Lifted Heuristics
Towards more scalable Planning Systems

Ridder, Bernardus

Awarding institution:
King's College London

Title: Lifted Heuristics

Towards more scalable Planning Systems

Author: Bernardus Ridder

Lifted Heuristics:
Towards more scalable Planning Systems

Bernardus Cornelis Ridder
PhD Thesis
King's College
Department of Informatics

January 9, 2014

Abstract

In this work we present a new lifted forward-chaining planning system which uses new heuristics and introduces novel pruning techniques which can solve problem instances which - until now - cannot be solved by contemporary planners due to grounding.

State-of-the-art planning systems rely on grounding, enumerating all possible actions, before search can begin. Grounding is a necessary step for these planners because their domain analysis, heuristic computation, pruning strategies and even search strategies need this information as a prerequisite. This grounding step is an essential step for most (if not all) state-of-the-art planning systems. A few planning systems use lazy evaluation which means that an action is only grounded when it is needed. But in these strategies, for most domains, the set of actions that need to be grounded are all the actions so this does not solve the underlying problem.

This thesis presents two new heuristics - called lifted relaxed planning graph heuristic and lifted causal graph heuristic - that do not require the planning domain to be grounded. This makes our planning system applicable to larger problem instances because we have smaller memory constraints compared to state-of-the-art forward chaining planners. Heuristics have been presented in the past which did not require grounding (for example least-commitment planners like Partial-Order Planners), but the weakness of their heuristics prevents them to compete with the state of the art. The heuristics presented in this thesis compare favourably to the state-of-the-art.

We build on previous work done on symmetry breaking in order to abstract the planning problem and prune the search space. Symmetry relationships explored in the past are quite restrictive and are only useful in problems which are highly symmetrical. We relax this definition and build upon almost symmetry which finds more symmetrical relationships and allows us to construct the data structures like the lifted relaxed planning graph and lifted transition graph using less memory and time.

In this work we present a new lifted forward-chaining planning system which uses new heuristics and introduces novel pruning techniques which can solve problem instances which - until now - cannot be solved by contemporary planners due to grounding.

State-of-the-art planning systems rely on grounding, enumerating all possible actions, before search can begin. Grounding is a necessary step for these planners because their domain analysis, heuristic computation, pruning strategies and even search strategies need this information as a prerequisite. This grounding step is an essential step for most (if not all) state-of-the-art planning systems. A few planning systems use lazy evaluation which means that an action is only grounded when it is needed. But in these strategies, for most domains, the set of actions that need to be grounded are all the actions so this does not solve the underlying problem.

This thesis presents two new heuristics - called lifted relaxed planning graph heuristic and lifted causal graph heuristic - that do not require the planning domain to be grounded. This makes our planning system applicable to larger problem instances because we have smaller memory constraints compared to state-of-the-art forward chaining planners. Heuristics have been presented in the past which did not require grounding (for example least-commitment planners like Partial-Order Planners), but the weakness of their heuristics prevents them to compete with the state of the art. The heuristics

presented in this thesis compare favourably to the state-of-the-art.

We build on previous work done on symmetry breaking in order to abstract the planning problem and prune the search space. Symmetry relationships explored in the past are quite restrictive and are only useful in problems which are highly symmetrical. We relax this definition and build upon almost symmetry which finds more symmetrical relationships and allows us to construct the data structures like the lifted relaxed planning graph and lifted transition graph using less memory and time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Planning – as a subfield of AI – is becoming increasingly more relevant to real-world applications because of the evolution of modeling languages and algorithms used to solve problems described in the modeling languages. Early languages like STRIPS model a real world problem as a set of formulae over grounded and function free first order predicates. These early languages do not model numbers and time. Subsequent languages that relax the constraints of the STRIPS language were introduced. ADL is such an example, this language allows negative preconditions and disjunctions over the goal literals. Modern modeling languages like PDDL can model time, numbers, continuous functions and many more features. These features are all important to model real world problems like subvoltage stations [45].

Planning systems have evolved too. For a long time research focused on least commitment planners, like *partial-ordered planners* [44] [59] [62]. These systems were seen as very flexible – as the name implies – choices could be made on the ordering of actions, bindings of variable and which subgoal to work on. However, ultimately these systems were not able to compete with forward chaining planners which started with the introduction of FF [32]. The problem with least commitment planners is twofold. First of all lacking an explicit state makes it hard to calculate an informative heuristic. Some attempts have been made to port heuristics from forward chaining planners to partial-ordered planners [42], but ultimately least commitment planners have fallen out of favour. Secondly, forward state planners have a limited search space, the search space for least commitment planners like partial ordered planners is not bounded.

We investigate a limitation that is shared by many – if not all – state-of-the-art planners, *grounding*. State-of-the-art planning systems rely on *grounding*, enumerating all possible actions, before search can begin. Grounding is a necessary step for these planners because their *domain analysis*, heuristic computation, pruning strategies and / or search strategies need this information as a prerequisite. For example, in order to construct a *relaxed planning graph* [32], *domain transition graph* [26] or *transition graph* [28] we need to ground all the action in a *typed planning problem*. Grounding is a powerful technique because – among other reasons – it allows the computation of more *informative* heuristics. Heuristics that do not require grounding cannot compete with the current state-of-the-art methods. Examples include heuristics developed for

least commitment planners, such as *partial-order planners* [44] [59] [62].

Despite the benefits of grounding there is a serious drawback to this enumeration – the amount of memory required to store all the grounded actions. This drawback is not noticeable when we try to find plans for the benchmark domains of the *international planning competitions* because the sizes of these domains are relatively small compared to *real-life* problems. *Real-life* problems are not necessarily more difficult than the problems presented in the benchmark domains but the size of the domains can be much bigger [14]. This means that, despite their strengths, state-of-the-art planners cannot even start solving these problems.

Attempts have been made in the past to either reduce grounding or forego it altogether, especially in Partial-Order planners such as NONLIN [59], UCPOP [44] and VHPOP [62]. As stated before, these approaches cannot compete with state-of-the-art approaches which rely on grounding. The scalability of the current best approaches benefit from highly effective heuristics but are limited by the size of problems they can ground. Although not as competitive, the lifted approaches are not limited in this way. So there is great untapped potential to solve larger problems if only we could get good heuristic guidance.

A possible solution to this problem is *symmetry breaking*. Planning problems like *Gripper* contain many objects, but most objects have the same initial state and the same goal that needs to be achieved. So instead of exploring all possible actions that can be applied to a state, we limit ourselves to only those actions that are *functionally* citesymmetry1999 different. Breaking symmetry is utitilised in many CSPs and SAT solvers [55] but is not as prevalent in planning systems. Earlier work demonstrated how *functional symmetry* can de detected and exploited to prune parts of the search space. However, while this technique works well on domains like *Gripper* which exhibits a lot of symmetry, there are many domains that exhibits very little or no functional symmetry.

The solution we propose in this thesis exploits *almost symmetry* [49] which detects more symmetrical relationships between *objects* than functional symmetry which can be treated *equivalently*. Almost symmetry relaxes the definition of functional symmetry, it does not require objects to have similar initial state and goals specified. Instead we detect almost symmetry relationships between objects that can become functionally symmetrical if we relax the planning problem. Based on this symmetry breaking we can *abstract* the planning problem and treat sets of objects as *equivalent*.

The planning system we describe in this thesis is a forward-chaining planner that does *not* require the domain to be grounded. We present two heuristics that are adaptations of the Fast Forward [30] and Causal Graph [26] heuristics, which are more informative than other heuristics that do not require grounding. The implementation of both heuristics introduced interesting challenges which are discussed and solved in this thesis. Apart from the *lifted* heuristics, we present novel pruning techniques.

This thesis is structured as follows:

In Chapter 2 we introduce the relevant background. The first section describes the modelling languages that have been and are used to model planning problems. Next we discuss all the heuristics (and planning systems which make use of them) that are related to the heuristics and pruning techniques we introduce for forward-chaining planners. To understand alternative searching systems which do not require

grounding we discuss least commitment planners in the section after that. Next we present techniques which have been developed to translate PDDL (Planning Domain Definition Language) problems into SAS^+ problems and perform domain analysis. TIM is one of these techniques and is used in the subsequent two chapters to carry out domain analysis. Finally we discuss previous methods that have been developed to do *symmetry breaking*.

In Chapter 3 we present the planning system we have written, inspired by the Fast Forward planner. It is a forward-chaining planner that abstracts the planning problem by exploiting *almost symmetry* relationships between objects. We show how we find these equivalence relationships between objects and how we use this information to abstract the planning problem. By using these equivalence relationships we can calculate the *lifted relaxed planning graph* heuristic that compares favourably to the Fast Downward heuristic, and which calculates more quickly and uses less memory. This means this planning system can solve larger problem instances than Fast Forward can. We introduce a novel pruning technique and demonstrate several alternative implementations of the lifted relaxed planning graph heuristic. Finally we show the results, where we compare: time, states explored, plan quality, and memory usage. Finally we show a domain that cannot be solved by state-of-the-art planners, but which are solvable by our planning system.

In Chapter 4 we introduce our second heuristic, called the lifted causal graph heuristic. This is a variation of the causal graph heuristic used by Fast Downward. We show how we use the domain analysis performed by TIM in order to construct the *lifted transition graphs* – which serve the same purpose as the *domain transition graphs* – and the *causal graph*. After these structures are defined we show how we utilise TIM's analysis by *merging* lifted transition graphs to reduce the number of cycles in the causal graph without having to remove any information from the planning problem. This merge step has been inspired by the *Merge and Shrink* algorithm and is a novel contribution for calculating this heuristic. We use the *equivalence* relationships between objects, as introduced in the previous section, to partially ground the lifted transition graph structures. However, unlike the domain transition graphs we do not end up with fully lifted atoms and actions. Finally we show how the lifted causal graph is calculated and compare the results with the causal graph heuristic, the additive enhanced context heuristic and the merge and shrink heuristic. Our approach uses less memory and is able to calculate the heuristics of states more quickly without having to sacrifice the informativeness of the heuristics.

Finally Chapter 5 describes our conclusions and introduces ideas and direction for future work.

Chapter 2

Background

2.1 Modelling Languages

A planning problem is the task of finding a sequence of actions to take us from the initial state to a goal state. One of the first languages used to model planning problems was *STRIPS* (Stanford Research Institute Problem Solver) [13]. With STRIPS a planning problem is modelled with first-order logic formulae, where the literals are grounded and function free. A state is defined as a conjunction of positive grounded literals. STRIPS assumes a closed world, which means that any literal that is not part of a state is assumed to be false. A goal is defined as a conjunction of literals. Actions contain preconditions and effects. The preconditions are a set of positive literals which need to be true in a state before the action can be applied to that state. Effects are a set of literals; each literal in this set can either be positive or negative. When an action is applied to a state all the negative effects are removed from the state before the positive effects are added; this function constructs the successor state. A solution to the planning problem is a sequence of actions that can be applied, starting from the initial state, and the resulting final state is a superset of the goal.

Additional languages were developed after STRIPS which relaxed some of the limitations of STRIPS. For example, *ADL* (Action Description Language) [43] allows both positive and negative literals in states. A goal can either be a conjunction or a disjunction over literals. In addition it introduced new language features, for example quantified variables in goals, conditional effects, and typed variables. Eventually the *de facto* language for modelling planning problems became *PDDL* (Planning Domain Definition Language) [22]. This language was developed for the first *International Planning Competition* (IPC) in 1998 and, with every competition since, the language has evolved to incorporate new features. PDDL splits the definition of a planning task into two parts: (1) A *domain description* specifies all elements which apply to every possible problem instance for that domain. It specifies the PDDL features used to model the problem task, including the type hierarchy, constant objects, predicates, and operators. (2) A *problem description* defines the actual problem we try to solve and specifies the objects and their types, the initial state, and the goal state.

The different variants of PDDL are listed below:

- PDDL 1.2 was the language used in the first two IPCs in 1998 and 2000.
- PDDL 2.1 was the language used in the third IPC in 2002; it introduced numeric fluents, plan-metrics, and durative/continuous actions.
- PDDL 2.2 was the language used in the forth IPC in 2004; it introduced derived predicates and timed initial literals.
- PDDL 3.0 was the language used in the fifth IPC in 2006; it introduced state-trajectory constraints and preferences or soft goals.
- PDDL 3.1 was the language used in the sixth and seventh IPCs in 2008 and 2011 respectively. It introduced objects-fluents.

2.2 Classical Planning

In this work we focus on *classical planning* problems, i.e. planning problems which are *deterministic*, *fully observable*, and do not deal with numbers or time. We slightly extend the definition of classical planning problems to bring it in line with the PDDL 1.2 language by including a type hierarchy. In addition, we use a more general definition for operators and atoms; this allows us to use the same definition in later chapters when we discuss the *lifted* heuristics.

Definition 1 — Typed planning task
A typed planning task is a tuple $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ where:

- T is a set of types. Every type $t \in T$ has a set of super types, written $SuperType(t)$.
- O is a set of objects, each object $o \in O$ is associated with a type $t \in T$, written $Type(o)$.
- P is a set of predicates, where a predicate $p \in P$ is a tuple $\langle name, types \rangle$. $name$ is a sequence of characters and $types$ is a sequence of *types*.
- A is a set of operators, where an operator $a \in A$ is a tuple $\langle name, parameters, precs, effects \rangle$, $parameters$ is a sequence of variables. $precs$ and $effects$ are sets of atoms. An atom is a tuple $\langle p, V \rangle$, where $p \in P$ and V is a sequence of variables. A variable v is a pair $\langle t, D_v \rangle$, where $t \in T$ and $D_v \subseteq O$ is the domain. We refer to the ith variable with the notation V_i. If the size of all domains of all variables of an atom is exactly one we call that atom *grounded*. Likewise, if the size of all the domains of the $parameters$ is exactly one we call the action *grounded*. We use the notations $effects^+$ and $effects^-$ for the subset of atoms in $effects$ that are positive and negative, respectively.
- s_0 is set of grounded atoms called the initial state.

- s_g is set of grounded atoms called the goal.

Definition 2 — Applying a grounded action

Given a state s and a grounded action $a \in A$ we define the result of applying a to s as follows:

$$Result(a, s) = \begin{cases} s \setminus a_{effects^-} \cup a_{effects^+} & a_{precs} \subseteq s \\ undefined & otherwise \end{cases} \tag{2.1}$$

When the effects of applying an action to a state is defined we say that that action is applicable to that state. The result of applying a sequence of grounded actions is defined as:

$Result(\langle a_0, a_1, \ldots, a_n \rangle, s) = Result(a_n, Result(\langle a_0, a_1, \ldots, a_{n-1} \rangle, s))$.

Definition 3 — Solution to a typed planning task

Given a typed planning task $\Pi = \langle T, O, P, A, s_0, s_g \rangle$, a solution is a sequence of grounded actions $\phi = \{a_0, a_1, \ldots, a_n\}$, such that $s_g \subseteq Result(\phi, s_0)$. We call $\mid \phi \mid$ the quality of the found solution, and *iff* there exists no other solution ϕ' for Π such that $\mid \phi' \mid < \mid \phi \mid$ we say that ϕ is the optimal solution.

In the AI literature there are many search techniques used to solve problems which are deterministic and fully observable. Instead of writing specialised planners to solve these planning problems one might just use A*, breath-first search, depth-first search, or any other general search techniques, and indeed many have. However, classical planning problems are *EXPSPACE-complete* [11], which means that the search space can be too big for these techniques to find a solution in a reasonable amount of time.

2.3 State-based planning

One of the search techniques developed to solve planning problems is to explicitly explore the search space. Search can start at the initial state and the search space is explored by generating successor states from any state that has been generated until a goal state is found. This method is called *Forward-Chaining Planning*. This method can be used with any other existing search strategy (breath-first search, depth-first search, etc) to navigate the search space. Currently this technique is utilised by most competitive planners.

Alternatively, search can start at the goal state and the search space can be explored by inverting the action definitions. The search stops when we find a state that is equal to the initial state. This search method is called *Backward-Chaining planning*. Instead of starting at the initial state to start planning we start at the goal state and work our way backwards. The rules for when actions are applicable to a state change as follows.

Definition 4 — Applying a grounded action

Given a (partial-)state S and a grounded action $a \in A$ we define the result of applying a to S as follows:

$$Result(a, S) = \begin{cases} S \setminus a_{effects^-} \cup a_{effects^+} & a_{precs} \subseteq S \\ undefined & otherwise \end{cases} \tag{2.2}$$

When the result of applying an action to a state is defined we say that that action is applicable to that state. The result of applying a sequence of grounded actions is defined as:

$Result(\langle a_0, a_1, \ldots, a_n \rangle, S) = Result(a_n, Result(\langle a_0, a_1, \ldots, a_{n-1} \rangle, S))$.

A solution to the planning problem still follows the same definition as for forward-chaining planning when we reverse the order of the found sequence of actions. For some problems backward-chaining has proven to be quite effective, but it depends on the structure of the problem. In general forward-chaining planning seems to be more robust, because the search space explored by backward search contains states that are not reachable from the initial state. In addition, whereas forward-chaining explores a search space that consists of complete state representations, backward-chaining explores a search space consisting of partially defined states. This is because – starting with the goal state – a state is actually a set of possible states.

2.4 Heuristics

In this work we are concerned with heuristics that are defined as follows. Given a typed planning task Π, a heuristic is a function that takes a state s as its parameter and estimates the minimal number of actions that need to be applied to s in order to reach a state that satisfies Π_{s_g}. The optimal heuristic h^* returns the minimal number of actions that need to be applied to s to reach a state that satisfies Π_{s_g}.

Heuristics can be separated into two categories: heuristics which are *admissible* return a value that is equal or less than the optimal heuristic h^* for any state. Admissibility is a desirable property if we want to find an *optimal* solution to a problem; we say that a solution is optimal if there exists no other solutions to a problem which contain less actions. If we use an admissible heuristic with a complete non-greedy search algorithm such as A^* then we are guaranteed that the first solution found is an optimal solution.

If a heuristic is not admissible no guarantee of optimality can be given. These heuristics can be used when we are not interested in finding optimal solutions but want to find out if a solution exists for a planning problem. In general the problem of plan existence is not easier than finding the optimal solution (e.g. there might only be a single solution), but for most planning problems finding *a* plan is easier than finding the optimal plan. In this section we will describe the most important heuristics that have been used and developed to solve planning problems, and provide examples of planning systems that make use of them.

In order to find a heuristic the original problem is *relaxed* such that finding a solution to the relaxed problem is tractable but also *informative*. The closer the relaxation is to the original problem the closer will the derived heuristic value be to h^*. There is, however, a trade-off. In general the more informative the heuristic, the more time it takes to calculate it. The extreme case is to apply no *relaxation* at all and solve the

original planning problem, which will yield the same heuristic as h^*. However, finding this heuristic is as hard as solving the problem itself. On the other hand, returning a 1 as the heuristic estimate for any state that does not satisfy the goal and 0 for states that do is very cheap to calculate but is not informative.

We shall discuss two types of relaxation that have been explored to construct heuristics. The first is to relax the delete effects of an action. This relaxation ignores all the delete effects of all the actions; this means that once a fact has been made true it can never be made false. This relaxation alone is, however, not enough to calculate a heuristic in a reasonable amount of time. As we shall discuss when we describe the h^{ff} heuristic, finding an optimal solution to the relaxed planning problem is NP-hard [5]. Therefore, heuristics using this relaxation apply further relaxations or policies to reduce the time needed to solve the relaxed problem.

Another way of relaxing the problem is by applying *abstractions*, a more general form of relaxations where any set of goals, preconditions and effects is ignored. Abstractions are used by hierarchical problem solvers which abstract a problem such that a solution is found by solving an hierarchy of abstracted problems. Initially a solution is found for the highest abstraction – which contains the least number of preconditions, effects and goals – and subsequently this solution is refined for the next abstraction. This process of refining solutions continues until a solution is found for the original planning problem. It is hard to find an abstraction hierarchy that is easier to solve than the original problem [1]. However, if we relax the requirement that the solution found by solving an hierarchy of abstractions is a solution to the original planning problem we can use the plan length of the found solution as a heuristic estimate. This method is used by the h^{cg}, h^{cea} and h^{ms} heuristics amongst other heuristics.

2.4.1 Heuristics based on the delete relaxation

In this section we will discus heuristics that are based on the relaxation that all the delete effects are removed from all the actions. As discussed before, finding an optimal solution for the relaxed plan is NP-hard. Therefore, all the heuristics introduced here apply further relaxations such that finding a plan is tractable

The add heuristic

The *add* heuristic(h^{add}) uses the delete relaxation to find a heuristic. In order to find a *plan* in polynomial time the h^{add} heuristic further simplifies the problem by assuming goal independence. This means that the actions added to the plan do not interact with each other, so a separate plan is found for every goal (and subgoal) independently. This is not an optimal solution to the relaxed plan, which renders this heuristic not admissible. The HSP [4] planner uses (amongst others) this heuristic with a weighted A* search algorithm to search the search space. The add heuristic is formally defined as:

Definition 5 — h^{add}

Given a planning problem $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ and a state s, the h^{add} heuristic is defined as:

$$h^{add}(s) = \Sigma_{l \in s_g} cost(s, l) \tag{2.3}$$

$$cost(s,l) = \begin{cases} 0 & \text{if } l \in s \\ 1 + \min_{a \in A | l \in a_{effects}} \Sigma_{p \in a_{precs}} cost(s,p) & \text{if } \exists_{a \in A} l \in a_{effects} \\ \infty & \text{otherwise} \end{cases} \tag{2.4}$$

The weakness of this heuristic is that apart from ignoring the delete effects it also discounts all the positive interactions between actions, e.g. if an action achieves n goals it will be included n times in the relaxed plan generated by h^{add}, where n is the number of goals.

The max heuristic

The *max* heuristic searches for a plan in the relaxed planning problem which achieves the most expensive (sub)goal. Unlike h^{add} it does not discount the positive interactions between actions. Instead it assumes that if the most expensive (sub)goal is achieved then the other (sub)goals are achieved as well by that action (or by previous actions). Unfortunately this heuristic has proven to be not very informative.

Definition 6 — h^{max}

Given a planning problem $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ and a state s, the h^{max} heuristic is defined as:

$$h^{max}(s) = max_{l \in s_g} cost(s, l) \tag{2.5}$$

$$cost(s,l) = \begin{cases} 0 & \text{if } l \in s \\ 1 + \min_{a \in A | l \in a_{effects}} max_{p \in a_{precs}} cost(s,p) & \text{if } \exists_{a \in A} l \in a_{effects} \\ \infty & \text{otherwise} \end{cases} \tag{2.6}$$

Critical path analysis

The critical path analysis searches for a plan which achieves the most expensive goal. Unlike h^{add} it does not discount the positive interactions between actions. Instead it assumes that if the most expensive (sub)goal is achieved then the other (sub)goals are achieved as well by that action (or by previous actions). This algorithm can be generalised by not only searching for a plan for the most expensive (sub)goal but for the m most expensive (sub)goals. These are members of the $h^m \mid m \in \mathbb{N}_1$ family of heuristics where $h^{max} = h^1$ [24]. The h^m heuristic tries to find the cost of the most costly subset of goals of size m.

Definition 7 — h^m

Given a planning problem $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ and a state s, the h^m heuristic is defined as:

$$h^m(s) = \max_{L \in C^m(s_g)} cost(s, L) \tag{2.7}$$

$$cost(s, L) = \begin{cases} 0 & \text{if } L \subseteq s \\ 1 + \min_{a \subseteq A | L \subseteq \bigcup_{a_{effects}}} \max_{L' \in C^m(a_{precs})} cost(s, L') & \text{if } \exists_{a \subseteq A} L \subseteq \bigcup_{a_{effects}} \\ \infty & \text{otherwise} \end{cases} \tag{2.8}$$

where $C^m(L)$ are all possible sets of literals of size m given a set of literals L. If the size of the set of literals L is equal to or less than m, then $C^m(L)$ is equal to L.

The FF heuristic

A more informative heuristic than the h^{add} heuristic is the so-called FF heuristic, or h^{ff}. The name comes from the planner that first utilised it, Fast Forward. The reason h^{ff} is more informative than h^{add} is because it takes positive interaction between actions into account. To compute the FF heuristic a *Relaxed Planning Graph* (RPG) is constructed, which is based on the *planning graph* [3]. An RPG is different from a planning graph because the delete effects of an action are ignored. This means that there are no *mutex* relationships between actions and facts.

Definition 8 — Relaxed Planning Graph

Given a typed planning graph $\Pi = \langle T, O, P, A, s_0, s_g \rangle$, an RPG is a tuple $rpg_\Pi = \langle \mathit{fl}, al \rangle$, which consists of alternating fact layers (fl) and action layers (al). In order to construct the RPG we augment the set of actions, A; for every possible grounded atom l we add a *NOOP* action to A whose precondition and effect is l. Then rpg is constructed as follows. The first fact layer fl_0 consists of the set of literals which are true in s_0, then for $i \in \mathbb{N}_1$:

$$al_{i-1} = A' \subseteq A \mid \forall_{a \in A'} a_{precs} \subseteq \mathit{fl}_{i-1} \tag{2.9a}$$

$$\mathit{fl}_i = \mathit{fl}_{i-1} \bigcup_{a \in al_{i-1}} a_{effects^+} \tag{2.9b}$$

When two consecutive fact layers have the same set of facts we know that no other facts can be made true and we say we have reached the *level off point*. Any fact that is not present in the last fact layer at level off point is unreachable. So if a literal $l \in s_g$ is not present in the last fact layer we know the planning task is unsolvable. The reverse, however, is not true. Any fact which appears in the last fact layer is not guaranteed to be reachable in the actual problem. The reason is that the RPG tries to solve a relaxed version of the original planning problem.

After the RPG is constructed a relaxed plan is extracted. The length of the relaxed plan is the heuristic estimate for the original problem.

Definition 9 — Relaxed Plan

Given a typed planning task $\Pi = \langle T, O, P, A, s_0, s_g \rangle$, a sequence of set of actions $\phi = \{A_1, \ldots, A_{|al|}\}$ and an RPG $rpg_\Pi = \langle \mathit{fl}, al \rangle$, we define the supported facts as:

$$F_i = \begin{cases} \mathit{fl}_0 & \text{if } i = 0 \\ F_{i-1} \bigcup_{a \in A_i} e \in a^+_{effects} & \text{if } \forall_{a \in A_i} a_{precs} \subseteq F_{i-1} \\ \text{undefined} & \text{otherwise} \end{cases} \tag{2.10}$$

ϕ is a *relaxed plan iff* $\forall_{i \in \{0,\ldots,|al|-1\}} F_i$ is defined and $s_g \subseteq F_{|\mathit{fl}|-1}$. Informally, every set of actions $A_i \in \phi$ is a subset of the actions in the corresponding fact layer fl_i, such that all its preconditions are a subset of the supporting facts F_i. Finally we make sure that all the facts in the goal s_g are achieved by the relaxed plan.

Definition 10 — h^{ff}

Given a typed planning task Π, an RPG $rpg_\Pi = \langle \mathit{fl}, al \rangle$ and a relaxed plan $\phi = \{A_0, \ldots, A_{|al|}\}$, we define the FF heuristic as:

$$h^{ff} = \Sigma_{x=0}^{|al|} \mid A_x \mid \tag{2.11}$$

Because the RPG is a relaxed version of the original plan the h^{ff} heuristic is admissible if we find the optimal solution, but unfortunately finding an optimal solution of this planning problem is NP-hard [5]. Fortunately finding *a* relaxed plan can be achieved in polynomial time[5].

Given a planning problem Π, Fast Forward constructs the RPG until a fact layer fl_i is constructed which contains all the literals which appear in the goal Π_{s_g}. All the literals which appear in the goal Π_{s_g} are added to an open list L. For each literal $l \in L$ Fast Forward searches for the earliest fact layer $\mathit{fl}_j \mid \neg\exists_{k<j} l \in \mathit{fl}_k$ such that $l \in \mathit{fl}_j$. Next an action $a \in al_{j-1} \mid l \in a_{effects^+}$ that achieves l is selected from the preceding action layer. If more than a single action can achieve l then the action whose *action cost* is lowest is added to the relaxed plan.

Definition 11 — Action Cost

Given an action a in action layer al_i the *action const* is defined as:

$$actioncost(a) = \Sigma_{p \in a_{precs}} cost(p, i), \tag{2.12}$$

where $cost(p, i)$ is defined as

$$cost(p, i) = \begin{cases} 0 & \text{if } i = 0 \\ i & \text{if } NOOP \notin achievers(p) \\ cost(p, i-1) & \text{otherwise} \end{cases} \tag{2.13}$$

After an action is selected to achieve l we add its preconditions to L and remove l from L. We repeat this process until L is empty. h^{ff} is then equal to the number of actions in the relaxed plan.

Although h^{ff} is more informative than h^{add}, its heuristic alone is not the reason why Fast Forward performed so well in the international planning competitions. Recent work shows that near perfect heuristics still require an exponential amount of time as the task size grows larger [29]. This suggests that other techniques, such as pruning, are necessary to cope with larger problems. Fast Forward has implemented an incomplete search algorithm called *Enforced Hill-climbing* that prunes the search space quite aggressively. Starting with the initial state s_0 it performs a breath-first search until it reaches a state s whose heuristic value is better. It then restarts the algorithm with s as the current state and continues this process until it finds a goal state. Although this algorithm is incomplete it does allow Fast Forward to prune large parts of the search space.

Fast Forward prunes further by limiting the actions it considers to those that are *helpful*.

Definition 12 — Helpful Actions

Given an RPG and a relaxed plan, let L_1 be the set of literals achieved by the effects of all the operators in al_0 which are part of the relaxed plan.

An action $a \in A$ is *helpful iff* $a_{precs} \subseteq fl_0$ and $a_{effects} \cap L_1 \neq \emptyset$.

This pruning technique ignores all actions which are not relevant to reaching the goal state. For example, if we consider a mail delivering problem with the goal of delivering a single letter, we do not want to consider delivering any other letters because these are irrelevant to achieving the goal.

Because neither pruning techniques used by Fast Forward preserve completeness, it has a backup option in case the enforced hill-climbing with helpful action pruning fails. If no solution is found Fast Forward falls back on a *Best First* search algorithm, which is complete.

2.4.2 Heuristics based on abstractions

Relaxing a plan by removing all the delete effects from the actions is a subset of all possible *abstractions* that can be applied to relax a planning problem. We can make an abstraction of a planning problem by removing any preconditions and effects from the actions and by removing any of the atoms from the goal.

Definition 13 — Abstraction hierarchy

Given a planning problem $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ we define the function $level : a \rightarrow \mathbb{N}$ which maps every fact in s_g and every precondition and effect for all actions $a \in A$ to a natural number which denotes at which level this atom is included. We can make a hierarchy of abstractions by using a mapping function M_i that takes the original problem Π and maps it to the abstract problem at layer i that includes only those atoms a for which $level(a) \geq i$.

It is easy to see that the delete relaxation is a specific abstraction, where the mapping function maps every negative effect to $-\infty$. A hierarchy of abstractions is used by

planners like ABSTRIPS [56], ALPINE [36], and HIGHPOINT [1]. Given a mapping function M and a planning problem Π, these planners first find a solution for the most abstract planning problem $M_i(\Pi)$ and subsequently refine the found solution for the next abstraction $M_{i-1}(\Pi)$. This process continues until a solution is found for $M_0(\Pi)$, which is the original problem.

The idea behind this method is that it is easier to find a plan at a higher abstraction and then refine it than it is to find a plan from scratch. However, not every possible hierarchy of abstractions will make the overall planning problem easier. If a solution exists for a planning problem then an abstraction has the *upward solution property* [60] if a solution also exists for the abstraction. Not every abstraction that can be created using Definition 13 satisfies this basic property. For example, given a planning problem Π we could define an abstraction which removes all the positive effects that achieve any goal $g \in \Pi_{s_g}$. If there exists any fact $g \in \Pi_{s_g}$ such that $g \notin \Pi_{s_0}$ then this abstraction does not satisfy the upward solution property. The delete relaxation abstraction does satisfy this property, because we do not allow any negative preconditions or goals.

Previous work [36] has defined another important property called the *ordered monotonicity property*. This property ensures that a solution to an abstraction $M_i(\Pi)$ can be obtained by refining the solution of the abstraction $M_{i+1}(\Pi)$ without removing any of the actions from its plan and making sure that the actions in the abstract plan remain *relevant*. This does not mean that a refinement of an abstract plan for any $M_j(\Pi)$ is possible, so backtracking might still be necessary. It does mean that if any solution to $M_j(\Pi)$ can be obtained by refining the solution of $M_{j+1}(\Pi)$ then we do not need to remove any actions from the solution to $M_{j+1}(\Pi)$.

The *ordered monotonicity property* is in itself not a strong enough condition to guarantee that creating an hierarchy of abstractions and solving them is easier than solving the original problem. A stronger property called the *downward refinement property* [1] guarantees that any abstract solution can be refined to a concrete solution. This is a strong requirement and work [1] has shown that very few planning problems do not exhibit this property. Rather, given an abstract solution, the authors calculated a probability that the abstract solution can be refined.

Working example of an abstraction hierarchy

We shall now present a working example of how an abstraction can be obtained. This example is based on the HIGHPOINT [1] planning system. Abstraction layers are constructed based on the *order* constraints imposed on the atoms. Those atoms that are goals, or preconditions of actions that achieve goals, are higher up in the hierarchy than effects which help achieve the goal indirectly. Effects that are not relevant to achieving any goal will only be present in the lowest layer.

Definition 14 — Relevant effects

Given a planning problem $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ where all the atoms and actions are grounded, then we call every effect $e \in a_{effects} \mid a \in A$ relevant *iff*

- $e \in s_g$
- $\exists_{a' \in A} \exists_{e' \in a'_{effects}} e'$ is relevant $\wedge\, e \in a'_{precs}$

Definition 15 — Order constraints

Given a planning problem $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ where all the atoms and actions are grounded, then for each operator $a \in A \mid \exists e \in a_{effects} e$ is relevant we add the following order constraints:

- $Level(e) \geq Level(e') \mid e, e' \in a_{effects}$.
- $Level(e) \geq Level(p) \mid e \in a_{effects} \wedge p \in a_{precs}$.

This requires any relevant effect to be on the same or higher level than any other effect. In addition the preconditions are either on the same or a lower level than the effects of an action.

In order to build the hierarchies all the atoms in a planning problem are added to a directed graph. An edge exists from a to a' *iff* there exists an ordering constraint $a \leq a'$. Next, atoms are grouped together if they are part of the same strongly connected component. If there exists a path from a to a' and from a' to a then both atoms are part of the same strongly connected component. Each of these sets of atoms will be added to their own abstraction layer. The ordering of the abstraction layers depends on the dependencies between atoms in separate strongly connected components and multiple orderings might be possible.

Example 2.4.1 *Take a* towers of Hanoi *problem. The rules of the game prevent larger disks being put on smaller disks, which means that the largest disk needs to be put into position first. After the largest disk is in position it does not need to be altered any more. This problem is a good example of showing the benefits of constructing an abstraction hierarchy.*

In our example we use three pegs and three disks called SMALL, MEDIUM, LARGE. *The only action available is the following:*

- $MOVE\ d\ -\ disk\ from,\ to\ -\ peg$
- $preconditions$:
  - $(on\ d\ from)$
  - $(not\ (and\ (on\ SMALL\ from)\ (is-larger\ d\ SMALL)))$
  - $(not\ (and\ (on\ MEDIUM\ from)\ (is-larger\ d\ MEDIUM)))$
  - $(not\ (and\ (on\ LARGE\ from)\ (is-larger\ d\ LARGE)))$
  - $(not\ (and\ (on\ SMALL\ to)\ (is-larger\ d\ SMALL)))$
  - $(not\ (and\ (on\ MEDIUM\ to)\ (is-larger\ d\ MEDIUM)))$
  - $(not\ (and\ (on\ LARGE\ to)\ (is-larger\ d\ LARGE)))$
- $effects$:
  - $(not(on\ d\ from))$
  - $(on\ d\ to)$

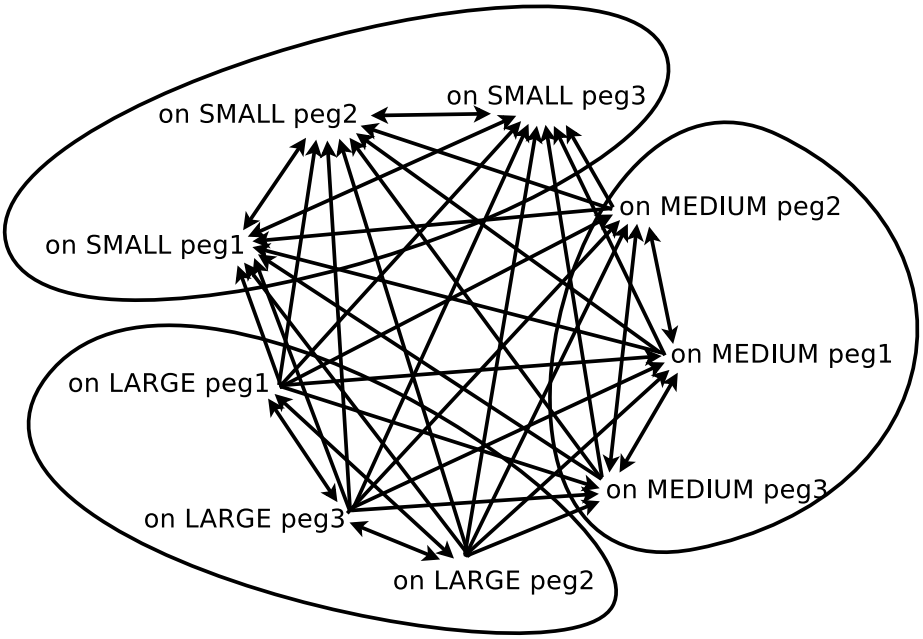

Figure 2.1: The order constraints between al the atoms for the towers of Hanoi problem.

The constructed directed graph is depicted in Figure 2.1; the circled atoms are the strongly connected components.

The algorithms and the ordering between the abstractions are depicted in Figure 2.2

This abstraction has both the *ordered monotonicity* and *downward refinement* properties. We start planning by solving the goal involving only the large disks. Then we refine this solution by solving the goals for the medium disks and lastly this solution is refined by including the small disks.

Causal Graph heuristic

Unfortunately, for most planning problems it is quite hard to find abstractions which satisfy both the *ordered monotonicity* and *downward refinement* properties. Without these properties there is no guarantee that a solution to an abstraction can be refined to a solution of the original planning problem without backtracking. However, if abstractions are used as a means to generate heuristic estimates instead of solutions to the original plans then the absence of these properties merely affects the quality of the found heuristic. This is the idea behind the causal graph heuristic (h^{cg}).

Instead of working directly on typed planning problem Π, Fast Downward transforms Π into a *typed multi-valued planning problem* which allows state variables to have domains that can contain a finite number of atoms. The modelling language PDDL encodes a state as a set of literals which can either be true or false. However, most literals in a planning problem are not independent of each other. If we find

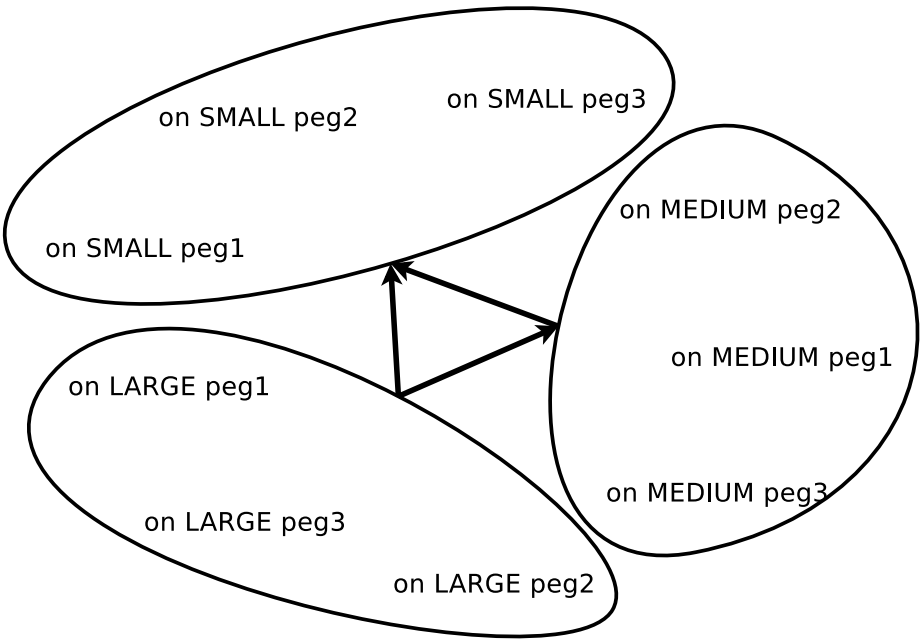

Figure 2.2: The abstractions derived from the towers of Hanoi problem.

a clique of literals which are all mutex with each other, such that only a single one of them can be true in any state that is reachable from Π_{s_0}, then we can encode this set of atoms more concisely. For example, in the *Driverlog* domain a truck can only be at a single location at any given time. The *Fast Downward* planner tries to find these mutex sets and re-encode the planning problem by introducing multi-valued state variables that can take any number of values instead of only true or false. So the state variable related to a truck might take as many values equal to the number of locations it can drive to. The formalism used is based on the SAS^+ planning model:

values equal to the number of locations it can drive to. The formalism used is based on the SAS^+ planning model:

Definition 16 — Typed Multi-Valued Planning Problem
A typed multi-valued planning task is a tuple $\Pi = \langle T, O, P, Q, A, s_0, s_g \rangle$ where:

- T is a set of types. Every type $t \in T$ has a set of supertypes, written $SuperType(t)$.
- O is a set of objects. Each object $o \in O$ is associated with a type $t \in T$, written $Type(o)$.
- P is a set of predicates, where a predicate $p \in P$ is a tuple $\langle name, types \rangle$. $name$ is a sequence of characters and $types$ is a sequence of *types*.
- Q is a finite set of state variables, each with an associated finite domain D_q. Each element in D_q is a grounded atom. An atom is a tuple $\langle p, V \rangle$, where $p \in P$ and

V is a sequence of variables. A variable v is a pair $\langle t, D_v \rangle$, where $t \in T$ and $D_v \subseteq O$ is the domain. We refer to the ith variable with the notation V_i. If the size of all domains of all variables of an atom is exactly one we call that atom *grounded*. A partial variable assignment or partial state over Q is a function s on some subset of Q such that $s(q) \in D_q$ wherever $s(q)$ is defined.

- A is a set of operators, where an operator $a \in A$ is a tuple $\langle name, parameters, precs, effects \rangle$. $parameters$ is a sequence of variables. $precs$ and $effects$ are partial assignments over $Q \mid \forall_{D_q \in Q} \forall_{\langle p, V \rangle \in D_q} V \subseteq parameters$. If the size of all the domains of the $parameters$ is exactly one we call the action *grounded*.
- s_0 is a state over Q called the initial state.
- s_g is a partial state over Q which satisfies the goal.

The definitions for when an action is applicable and what a solution to a planning problem is remain the same, except that the atoms are encapsulated in state variables. In the worst case scenario we cannot find any cliques of mutually exclusive grounded atoms, which means that the domains of all the state variables will be of size two: $\{\top, \bot\}$, where $\top$ means that the grounded atom is true and $\bot$ means that the grounded atom is false. Various methods have been developed to translate a typed planning problem into a typed multi-valued planning problem. Examples include TIM [15], DISCOPLAN [21], and the method used by Fast Downward [8]. These methods are described in more detail in Section 2.6.

The relationships between the values of a state variable are encoded in a *Domain Transition Graph* (DTG).

Definition 17 — Domain Transition Graph

Given a state variable $q \in Q$ a *Domain Transition Graph* is a labeled directed graph with vertex set D_q. The following edges are added:

- An edge is added between two vertices q_{from} and q_{to} for every grounded action $a \in A$ if $q_{from} \in a_{precs}$ and $q_{to} \in a_{effects}$, with label $a_{precs} \setminus q_{from}$.
- An edge between q_{to} and every edge $q_n \in (D_q \setminus q_{to})$ if $q_n \notin a_{precs}$ and $q_{to} \in a_{effects}$, with label a_{precs}.

A *DTG* records the transitions between the values of a state variable. The labels record the preconditions which are part of a different state variable. The dependencies between different state variables are captured in a *causal graph*.

Definition 18 — Causal Graph

Given that a typed multi-valued planning task is a tuple $\Pi = \langle T, O, P, Q, A, s_0, s_g \rangle$, a causal graph is a directed graph $CG(\Pi)$ where the set of state variables Q is the vertex set. An edge (v, v') exists *iff* $v \neq v'$ and one of the following conditions hold:

- The DTG of v' has a transition with some precondition on v.
- The set of affected variables in the effect list of some action $a \in A$ affects both v and v'.

Based on the dependencies in the causal graph we can create the hierarchy of abstractions. The *ordered monotonicity* or *downward refinement* properties depend on the structure of the causal graph. For example, if there are no edges in the causal graph it means that none of the state variables affect each other and are not dependent on each other, which means that both properties are satisfied. An observation made in the journal paper on Fast Downward [26] is that given a typed multi-valued planning problem Π, if $CG(\Pi)$ is acyclic and the DTG for every state variable in Π is strongly connected, then Π has a solution. The same paper presents an algorithm which can find a solution for this type of problem without having to backtrack. This means that the *ordered monotonicity property* and *downward refinement property* are both satisfied. To find a solution can still require an exponential amount of time, because the generated plans can still be exponentially long.

The problem is that the causal graph for most planning problems are cyclic and the DTGs are not fully connected. However, given a multi-valued planning task Π it is possible to *abstract* Π in such a way that the causal graph of the abstracted problem is acyclic. For example, given the subset of all the state variables $Q' \in \Pi_Q$ and all the state variables on which it is dependent, if all the preconditions, effects, and goals that include any state variable $q \notin Q'$ are removed, then we obtain an abstraction that satisfies the *ordered monotonicity property*. This method can be used to remove dependencies between state variables. Given two state variables $q \in \Pi_Q$ and $q' \in \Pi_Q$ that are dependent on each other (i.e. they are connected in the causal graph), if all the preconditions and effects from each action that contains a value from q are removed then the edges between q and q' are removed from the causal graph. While this method preserves the *ordered monotonicity property* it does not have the *downward refinement property* unless the DTGs are totally connected. This means that there is no guarantee that this abstraction can be refined without backtracking, and so finding a solution using this abstraction can be as hard as solving the original problem.

This seems counter-intuitive, because if a DTG contains only edges with empty labels it means that there are no dependencies on other state variables. This means that the cost of changing the value of a state variable from v to v' is the shortest path in the corresponding DTG. This would suggest that a planning problem where there are no cycles in the causal graph is easier to solve. However, finding a plan for a planning problem that has no no cycles in its causal graph, even if the planning problem only has unary operators, does not change the complexity of finding a plan: it is still *PSPACE-complete* [23].

Fast Downward overcomes this difficulty by decomposing the abstracted problem whose causal graph contains no cycles into a number of subtasks with limited interaction.

Definition 19 — $SAS^+ - 1$

A $SAS^+ - 1$ task is a typed multi-valued planning problem Π with a designated variable $q \in \Pi_Q$ such that $CG(\Pi)$ has an edge from q to all other variables $q' \in \Pi_Q$, and no other arcs. This variable q is called the *high-level* variable, whereas all other state variables are called *low-level* variables. A goal must be defined for the high-level variable, and goals must not be defined for the low-level variables.

If we restrict the problem to finding a plan in $SAS^+ - 1$ it is an NP-complete problem [25], except when the DTGs of the low-level variables are totally connected; if that is the case then this abstraction has the downward refinement property and a solution can be found in polynomial time. The high-level variable is the only state variable with non-empty labels in the DTG, thus is the only variable with external preconditions. In order to calculate a heuristic in polynomial time an incomplete algorithm is introduced that does not guarantee to find a solution, but those it does find are valid. The algorithm to solve an $SAS^+ - 1$ task is listed by Algorithm 1.

Algorithm 1: $SAS^+ - 1$ solving algorithm. Adapted from [26]

foreach $d \in D_q \mid q \in Q \wedge \neg\exists_{q' \in Q}(q, q') \in CG(\Pi)$ **do**
  if $d \in s_0$ **then** $plan(d) = \langle\rangle$;
  else $plan(d) =$ undefined;
$Queue \leftarrow D_q$;
while $\|Queue\| > 0$ **do**
  $d \in Queue \mid \neg\exists_{d' \in Queue} \|plan(d')\| < \|plan(d)\| \wedge d \neq d'$;
  $Queue \setminus d$;
  if $d \in s_g$ **then return** $plan(d)$;
  $s \leftarrow Result(plan(d), s_0)$;
  foreach $(d, d') \in D_q$ **do**
    $cond \leftarrow label(d, d')$;
    $\pi \leftarrow \langle\rangle$;
    foreach $c \in cond$ **do**
      $\pi' \leftarrow$ the shortest plan to make c true from s;
      if π' *is undefined* **then**
        $\pi \leftarrow$ undefined;
        Break;
      else $\pi \leftarrow \pi \cup \pi'$;
    $\pi \leftarrow plan(d) \cup \pi \cup (d, d')$;
    if $\|plan(d')\| > \|\pi\|$ **then** $plan(d') \leftarrow \pi$;

This algorithm is complete as long as the DTGs of the low-level variables are fully connected. The algorithm greedily tries to find the shortest plan to make all the low-level variables match those of the precondition of the action and sticks with that assignment of the variables. This could mean that we run into a dead end which could have been avoided by achieving the low-level variables in a different way, but once we find a plan to reach a value of the high-level variable we commit ourselves to the found plan sequence.

With Algorithm 1 in place we can now provide the full algorithm for finding a heuristic for a typed multi-valued planning problem whose causal graph is acyclic. The cost of changing the value of a state variable q from $d \in D_q$ to $d' \in D_q$ is defined as $cost_q(d, d')$:

- If q has no dependencies in the causal graph, $cost_q(d, d')$ is equal to the length of the shortest path in the corresponding DTG or ∞ if no path exists.

- Let V_q be the set consisting of q and all dependencies of q in the causal graph of Π. Let Π_q be the planning task induced by V_q except that the initial value of q is set to d and the goal value of q is set to d'.

- $cost_q(d, d') = \|\pi\|$, where π is the plan for Π_q found by the $SAS^+ - 1$ planning algorithm. Here all high-level transitions have a cost of 1 and low-level transitions of state variable q_l from d_l to d'_l have $cost_l(d_l, d'_l)$.

The heuristic is calculated by the sum over $cost_v(s_0(q), s_g(q))$ over all variables $q \in Q$ for which the goal s_g is defined.

Unfortunately, most planning problems have cyclic dependencies in the causal graphs. In order to remove cycles in the causal graph Fast Downward removes dependencies between state variables. Whenever an edge in the causal graph between the state variables v and v' is removed, we update the labels of the edges in the DTG of v so that every precondition which contains a value of v' is removed. In addition, any effects on the state variable v' are removed from the actions contained in the DTG of v.

Whenever an edge is removed from the causal graph we abstract the problem further, which means that we are 'throwing away' information. We want to limit the amount of information we remove so that our heuristic estimate is as informative as possible. In order to preserve as many elements of the original planning problem as possible Fast Downward uses the following algorithm to determine which edges to remove from the causal graph.

The first edges to remove from the causal graph are those which are not relevant to any of the goals. We mark any state variable for which a goal is defined. Next we mark any state variable v for which an edge v, v' exists in the causal graph, where v' is marked. We continue this process until no more state variables can be marked. All state variables which have not been marked are not relevant to the goal can be removed, including all edges that contain an unmarked state variable. Next Fast Downward uses a greedy algorithm which iteratively calculates a total order on all the variables that are part of a cyclic strongly connected component, the order is based on the number of actions which induce each state variable. The variable induced by the fewest actions is removed from consideration and all edges to that node are removed. This process repeats until the causal graph is acyclic. The idea behind this method is to remove edges in such a way that those which are relevant to the least number of actions are removed first

Pattern Databases heuristic

Unlike the previous planning systems and algorithms described, the pattern database heuristic calculates an admissible heuristic. To do so it creates a set of abstractions for a planning problem. The hierarchy used is flat, so unlike the causal graph heuristic it does not create an hierarchy of abstractions. The idea behind the heuristic is that it creates a set of abstractions of a planning problem and stores the optimal solutions of these abstracted planning problems in a database. Then when we try to solve a problem we do a lookup in the database to check which solutions (or patterns) in the database

are relevant to the state we are exploring, and combine the cached solutions in such a way that the estimate does not overestimates the optimal heuristic for that state.

This method has been used in areas other than planning, such as finding optimal vertex covers of a graph, sliding puzzles and 4-peg tower of Hanoi problems [12]. Previous work has explored *non-disjunctive* pattern databases. For example, previous work [38] uses pattern databases to find a heuristic estimate for solving a Rubik's cube problem. The original problem is split up into three abstractions:

- Only consider the corners of the cube.
- Only consider six edges and ignore the corners.
- Only consider the other six edges.

These abstractions can be solved optimally, so when confronted with a Rubik cube configuration we lookup the precomputed heuristic estimates of the three abstractions and select the maximum heuristic amongst them. Since the solutions of the abstractions are optimal we are guaranteed that this is an optimal solution. We cannot combine the precomputed heuristics, because the corners and edges are not independent from each other. Any action that changes the configuration of an edge will affect the other edges and / or corners.

To improve upon *non-disjunctive* pattern databases we want to find abstractions that can be combined such that the sum of their heuristic estimates is still admissible. These heuristics are called *disjunctive* pattern databases. Previous work created domain dependent abstractions for specific problems. The pattern databases heuristic [7] creates these abstractions automatically. Instead of creating a single abstraction hierarchy it creates a set of abstraction hierarchies where each hierarchy consist of a single abstraction.

Definition 20 — Pattern Database

Given a planning problem $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ and a function $level : a \rightarrow \mathbb{N}$ then the abstracted plan Π' only contains atoms $f \in \bigcup_{a_{precs}|a \in A} \bigcup_{a_{effects}|a \in A} \cup s_0 \cup s_g$ for which $level(f) \neq \infty$. Let A be all the abstracted states that can be reached by the planning problem defined by Π'. In that case a pattern database for Π' is defined as:

$$PDB(\Pi') = \{S, \phi(S, \Pi'_{s_g}) \mid S \in A\}$$

where $\phi(S, \Pi'_{s_g})$ is length of the optimal solution from S to Π'_{s_g}.

To be able to store all the solutions of an abstracted planning problem we must either choose very small abstractions or limit the size of the pattern database. Given a limited amount of memory and a limited number of abstractions that can be defined we are faced with a bin packing problem. We want to define as many abstractions as possible within the given memory constraints. To solve this problem a best-fit strategy is used.

Definition 21 — Independent abstraction

We call an abstraction ϕ *independent iff* there are no operators that affect atoms both present in ϕ and atoms which are not in ϕ.

Theorem 2.4.1 *If the planning problem is partitioned into sets of* independent abstractions *then pattern databases generated for these partitions are* disjoint.

This is because an operator only affects the set of atoms in the set of independent abstractions and an operator of the abstract planning space contributes one to the overall estimate only if it changes atoms in the available partitions. Therefore, by adding the plan lengths of different abstract spaces, each operator on each path is counted at most once.

The benefit of pattern databases is that – once they are calculated – heuristics are computed in constant time because the results are stored in databases. However, as the problem instances grow we see that the effectiveness of pattern databases decreases because, proportional to the planning problem, fewer patterns can be stored if the memory available remains constant.

Merge and Shrink heuristic

Merge and Shrink [28] abstracts the search space by merging states. It transforms the DTGs into transition graphs by adding all possible transitions to every node. Transitions which do not affect the state variable are added to every node and have that node as its begin and end point. Formally we define these *transition graphs* as follows:

Definition 22 — Transition Graph

A *transition graph* is a tuple $\langle S, L, A, s_0, S_g \rangle$, where S is a set of states, L is a finite set of transition labels, $A \subseteq S \times L \times S$ is a set of labeled transitions, $s_0 \in S$ is the initial state and $S_g \subseteq S$ is the set of goal states. A path from s_0 to an $s_g \in S_g$ following the transitions of T is a plan for T. A plan is optimal *iff* the length of the path is minimal.

Given a typed multi-valued planning problem Π we can construct a transition graph $T(\Pi)$. Its states are constructed by taking the Cartesian product of all possible state variable assignments. The graph has an vertex, labeled by $a \in \Pi_O$, from s to s' if a is applicable in s and $Result(a, s) = s'$. A solution to Π is a path through the graph. The size of the transition graph is the size of the search space which – except for some trivial problems – is too big to represent in memory. In order to derive a heuristic from the transition graph we need to apply an *abstraction*.

Definition 23 — Abstraction

An abstraction of a *transition graph* is a pair $\langle T', \alpha \rangle$, where $T' = \langle S', L', A', s'_0, S'_g \rangle$ and $\alpha : S \to S'$ is a function called the *abstract mapping*, such that $L' = L$, $\langle \alpha(s), l, \alpha(s') \rangle \in A'$ for all $\langle s, l, s' \rangle \in A$, $\alpha(s_0) = s'_0$, $\alpha(s_g) = s'_g$ for all $s_g \in S_g$.

If T' contains no other transitions or goal states in addition to those required by the above definition then A is a *homomorphism*. An abstraction is transitive. Any solution of an abstraction of the original problem is an admissible estimate of the cost

of achieving the original problem. This is because all the transitions are retained; any path in T' is also a path in T. The abstractions we are interested in are projections on state variables.

Definition 24 — Projection

Given a planning task $\Pi = \langle T, O, P, Q, A, s_0, s_g \rangle$ and $Q \subseteq Q$, a projection is a homomorphism on $T(\Pi)$ defined by a mapping α such that $\alpha(s) = \alpha(s')$ *iff* $s(q) = s'(q)$ for all $q \in Q$. If Q only contains a single state variable then we call it an atomic projection.

If we create an atomic projection for every state variable we can derive a heuristic by adapting the h^{max} and h^{add} heuristics. Find a path in each transition graph for which a goal is defined and take the maximum or sum of this number. This approach would lead to a very poor heuristic because we ignore any transitions in other state variables. As in the h^{ff}, h^{cg}, and h^{cea} heuristics, better heuristic guidance can be achieved when we take the interactions between state variables into account. Merge and Shrink achieves this by merging the transition graphs of state variables. The synchronized product of two transition graphs is defined as follows:

Definition 25 — Synchronized product

Given two transition graphs $T' = \langle S', L, A', s'_0, S'_g \rangle$ and $T'' = \langle S'', L, A'', s''_0, S''_g \rangle$, their synchronized product is defined as $T' \otimes T'' = \langle S, L, A, s_0, S_g \rangle$, where $S = S' \times S'', \langle (s', s''), l, (t', t'') \rangle \in A$ iff $\rangle s', l, t' \rangle \in A'$ and $\rangle s'', l, t'' \rangle \in A'', s_0 = (s'_0, s''_0), S_g = S'_g \times S''_g$, and $\alpha : S \to S$ is defined by $\alpha(s) = (\alpha'(s), \alpha''(s))$.

The synchronized product of two abstractions of transition graph T is itself an abstraction of T. Forming the synchronized product is an associative and commutative operation, modulo isomorphism of transition graphs.

The synchronized product of all atomic projections of a typed multi-valued planning problem is equal to the full transition graph $T(\Pi)$. Taking the product of two projections yields a better defined search space and the shortest path found will give us a better heuristic estimate. If we were to take the product of all atomic projections, the resulting transition graph is a bijection of the full search space, thus the minimal path is an optimal solution to the planning problem. However, memory constraints prevent this from being an option for any reasonably sized problem. So in order to reduce the space required to store the abstraction (and to find a minimal path in a reasonable amount of time) we shrink the abstraction space by merging states.

Merge and Shrink uses a linear strategy to merge projections. An atomic projection is chosen and all other atomic projections are merged with the former until only a single abstraction remains. Every time a merge is performed we check if the size of the abstraction is above a certain threshold. If it is the abstraction is shrunk to a certain size and merging continues. Other, non-linear, merging strategies are possible but do not affect the heuristics derived.

The following rules determine in which order atomic projections are chosen:

1. If possible, choose a variable from which there is an arc in the causal graph to one of the previously added state variables.

2. If there is no such variable, add a variable for which a goal value is defined.

The atomic projections which are never chosen are those which are not relevant to achieving the goals. If multiple candidates are available the candidate with the "highest level" according to the ordering criterion used by Fast Downward is chosen.

In order to *shrink* the size of the merged transition graph $T = \langle S, L, A, s_0, S_g \rangle$ we apply a number of homomorphisms. Each maps two abstract states s and s' to a new abstract state $\{s, s'\}$ while mapping all other states to themselves. The number of homomorphisms applied is equal to $\mid S \mid -N$ where N is an arbitrary number. When selecting an atomic projection π_a to be merged with the merged transition graph π_m, both are shrunk prior to merging until $\pi_a * \pi_m \leq N$.

Care needs to be taken when selecting which states to merge. For example, if a goal state and initial state were to be merged the heuristic estimate would always be 0. To prevent this from happening we look for states whose minimal distance from the initial state and goal state are the same. By merging these states we are less likely to introduce short cuts into the abstract space which lead to poor heuristics. Additionally, when selecting which states to merge with identical distances, we prefer to merge those whose combined distance is largest. The reason for this is that when an A^* search is performed, we will only ever expand nodes whose combined distance is equivalent to or less than the minimal path from the initial state to a goal state. Thus by combining states with a high combined distance we hope to merge states which are not relevant in finding the shortest path.

2.4.3 Heuristics based on Landmarks

A different take on finding heuristic estimates allowed [54] to win the International Planning Competition in 2008 [1]. Instead of finding a relaxed plan from the initial state to the goal state it searched for literals that have to be made true in any solution to the planning task.

Previous work [33] extended the idea of goal ordering. Given two goals A and B and the knowledge that we cannot reach a state where B is true from any state where A is true without deleting A, it is reasonable to achieve B before we achieve A. Instead of limiting this idea to the top level goals, the search was extended to find orderings between different literals that have to be true in any feasible plan.

2.4.4 Landmark detection

Unfortunately deciding if a literal is a landmark is a PSPACE-complete problem [51]. The following is a sufficient condition for a fact to be a landmark:

Definition 26 — Landmark

Consider a solvable typed multi-valued planning task $\Pi = \langle T, O, P, Q, A, s_0, s_g \rangle$, and an atom $l = \langle p, V \rangle$. Define $\Pi_L = \langle T, O, P, Q, A_L, s_0, s_g \rangle$, where $A_L = \{ \langle name, parameters, precs, effects \rangle \in A \mid l \notin effects \}$. If Π_L is unsolvable, then l is a landmark.

While the above technique will work to find landmarks, it is a costly operation, even for planning problems with no delete effects where finding the landmarks takes

[1] http://ipc.informatik.uni-freiburg.de/

polynomial time using an RPG. Better techniques have been developed to find landmarks [33, 37, 50, 51, 53, 54]. Depending on how landmarks are found, different ordering constraints can be enforced on landmarks.

Definition 27 — Landmark Ordering
Given two landmarks A and B the following orderings hold:

- *Natural ordering: iff* in any solution to the plan A is always achieved before B we say A is naturally ordered before B, denoted $A \rightarrow B$.
- *Necessary ordering: iff* in any solution to the plan A is always true one time step before B is made true, denoted $A \rightarrow_n B$.
- *Greedy-necessary ordering: iff* in any solution to the plan A is true one time step before the first time B is made true, denoted $A \rightarrow_{gn} B$.
- *Reasonable ordering: iff* in any solution to the plan where B is true in a state but A is not, B must first be made false before A can be made true, denoted $A \rightarrow_r B$.

The latter can also be used to perform goal ordering and prune parts of the search space that reaches a goal prematurely.

The goals are obvious landmarks in any planning problem. We can find other landmarks by checking the preconditions of all actions that achieve a goal. The intersection of these preconditions must necessarily also be a landmark and ordered before the goal landmarks. Because this order necessarily holds in any solution to the planning problem we call this type of ordering a *natural order*. Furthermore the set of landmarks found this way are a conjunctive set of landmarks; all of them must hold true in a state reached whilst executing any feasible plan. All preconditions taken together form a disjunctive set of landmarks of which at least one must be true. Further landmarks can be discovered by taking the intersection of the preconditions of all actions that can achieve any of the facts in the disjunctive set. We can iterate over this process until we cannot discover any new landmarks. While this method is sound it will only find a very restrictive set of landmarks.

More landmarks can be found by restricting our attention to the *first achievers* of a landmark l. Thus we want to exclude any actions that depend, either directly or indirectly, on l. We can approximate this set of achievers by constructing an RPG and find the first fact layer fl_i where l is included. Next we take the intersection of the preconditions of the actions $a \subseteq al_i$ in the previous action layer for which $l \in a_{effects}$. This is not a sound approach as we are underestimating the set of actions that can achieve l without requiring l to be achieved first. This is because we do not look at actions that are part of action layers $al_j \mid j > i$ which do not require l as a precondition. Therefore, all candidates are tested using Definition 26 and pruned if they turn out not to be landmarks.

A similar technique which produces sound landmarks works by constructing a relaxed planning graph RPG_l until the level-off point but any action $a \in A$ that adds l is excluded. The intersection of the preconditions of all achievers A_l in the graph for l is a landmark and does not need any additional checking. Landmarks which are found

by looking at the first achievers are ordered *greedy-necessarily* before l. Disjunctive landmarks are found by combining the preconditions from A_l whose predicate names and arity are identical. Any such set which includes preconditions from every member of A_l is added as a disjunctive landmark. The idea behind this is that landmark analysis is easy to thwart. A planning problem that requires a package to be delivered with only a single truck will yield landmarks that inform us how that truck should drive and that the package should be loaded and unloaded from that truck. However, if we add another truck this information is lost. So what we want to obtain is a disjunctive landmark which informs us that *a* truck should pickup and drop that package.

Given a singleton landmark l we can check the DTG that corresponds with the state variable $q \in Q \mid l \in D_q$. If every possible path from $l_0 \in D_q \mid l_0 \in s_0$ to l contains a variable l', then l' is a landmark that is naturally ordered before l.

Given the set of landmarks found we can combine them into a *landmark-generation tree* where the landmarks are the nodes and directed edges specify the ordering constraints of the landmarks.

2.4.5 Utilising landmarks

Given a set of landmarks and ordering between them there are various ways in which these can be exploited to solve planning problems.

Search control

One technique [33] uses a landmark-generation tree to restrict the goals visible to the planning system. The original goals are masked and only the earliest landmark is disclosed to the planner; this is a form of greedy search. Given a planning problem $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ and a set of landmarks L and ordering constraints, the goal is restricted to $\bigvee_{L_g \subseteq L \mid \forall_{l_g \in L_g} \nexists_{l \in L} l \rightarrow_* l_g}$. In other words the goal is defined as the disjunction over those landmarks that have no other landmarks ordered before them. After one of the landmarks is achieved, it is removed from the set L and the set of goals is updated until the last landmark, a conjunction of literals s_g, is achieved.

This has been implemented on top of FF and LPG [19]. While limiting the goal focuses the planner to greedily find a solution to achieve any of the landmarks, it will generally produce longer plans because it does not consider the impact that achieving a landmark greedily has on the overall plan. In some cases this might even lead the planner into dead ends for planning problems that are solvable.

Heuristics guidance

Another option considered is to use landmarks as heuristics, as effected by the planner LAMA which uses the number of landmarks that need to be achieved before reaching the goal as part of its heuristic. This is given as $h^l = n - m + k$, where n is the total number of landmarks, m the landmarks which have been *accepted* and k the number of landmarks that need to be achieved again. A landmark l is accepted in a state s if it is true in that state and all landmarks ordered before l are accepted in the predecessor state from which s was created. An accepted landmark stays accepted in all successor

states. An accepted landmark is required again if it is not true in s and there is another landmark l' that is not accepted and is part of an greedy-necessary ordering constraint $l \rightarrow_{gn} l'$. While counting landmarks works quite well in some domains, it can be used on top of any other heuristic such as h^{ff} or h^{cg}.

2.5 Partial-Order Planning

Another search technique, *Partial-Order Planning*, starts, like backward-chaining planning, at the goal state and works its way back. Unlike the previous techniques it does not enforce a strict ordering on the actions applied. In both chaining planning techniques we apply an action to a (partial-)state and generate a successor state and continue this process until we have a fully sequenced set of actions which we can apply from the initial state until we reach a state which satisfies the goal. One of the appealing qualities of partial-order planning is the least commitment principle. In the context of partial-order planning this means that we do not have to commit to a fixed action ordering and we can work on individual parts of the problem, e.g. we can focus on a subset of the goal and leave the rest till later. The search space in partial-order planning is not composed of states, but rather of partial plans.

Definition 28 — Partial Plan
A partial plan is a tuple: $\langle A, L, O, B \rangle$, where:

- A is a set of actions.
- L is a set of *causal links*. A causal link $a_i \xrightarrow{q} a_j$ denotes that the precondition q of action a_j is achieved by action a_i.
- O is a set of orderings between the actions A. Unlike the previous chaining planning techniques not all actions need to be ordered.
- B is a set of binding constraints on the action parameters. If all the actions in a typed planning problem are grounded then $B = \emptyset$.

Due to the fact that search spaces for partial-order planners do not consist of states but partial plans we need to change the definition of a *typed planning problem*. The initial state s_0 becomes a special action a_0 with no preconditions and all of the literals as effects. The goal a_∞ becomes an action G with the literals as preconditions and no effects. The initial partial plan is defined as: $\langle \{a_0, a_\infty\}, \emptyset, \{a_0 \prec a_\infty\}, \emptyset \rangle$.

Planning is carried out by selecting a partial plan from the search space and refining flaws in that partial plan. When a partial plan contains no flaws we have found a solution to the planning problem. Flaws in a partial plan can be *open conditions* and *threats*.

An open condition $\xrightarrow{q} a_j$ means that the precondition q of action a_j has not been achieved yet. This flaw can be resolved by either finding an existing action $a_i \in A$ that has an effect which can be unified with q, or adding a new action which can achieve q. In both cases we must make sure that (1) the new ordering constraint $\{a_i \prec a_j\}$ and

(2) bindings of the action parameters of a_i to unify with q, do not violate any of the existing constraints. The casual link $a_i \xrightarrow{q} a_j$ is added to the new partial plan.

A threat exists in a partial plan if it contains a causal link $a_i \xrightarrow{q} a_j$ and an action a_n that contains an effect which negates q and a_n can be ordered between a_i and a_j. There are three possible refinements which can be applied to resolve this flaw.

- Order a_n before a_i by adding a new ordering constraint $a_n \prec a_i$. This is called *demoting*.
- Order a_n after a_j by adding a new ordering constraint $a_j \prec a_n$. This is called *promoting*.
- Add bindings constraints to the parameters of a_n such that its effect no longer negates q. This is called *separation*. This is only possible if not all actions are grounded.

A simple search algorithm for solving partial-order planning problems is given in Algorithm 2.

Algorithm 2: Partial-Order Planning Algorithm.

```
U ← Initial Partial Plan;
while U ≠ ∅ do
    u ∈ U;
    U ← U \ u;
    if u has no flaws then
        return u;
    f ← a flaw in u;
    U ← U ∪ Refinements(u, f);
```

As can be seen from this algorithm there are two choices to make: first of all we need to select which partial plan we want to select and secondly which flaw to resolve. The former is decided by a heuristic; we want the partial plan which is closest to the goal. The latter can be decided by either static policies or by heuristics.

2.5.1 Heuristics

The informativeness of heuristics is the Achilles' heel for partial-order planners. Like state-based planners, heuristics are used to determine how far a partial plan is from the goal. However, unlike state-based planners where we have to estimate how many more actions need to be executed before we reach a goal state, we have to take order and bindings constraints into account. Furthermore we de not have access to an explicit state description, which complicates matters further.

This is reflected in the heuristics used in planners such as UCPOP [44], which uses the number of flaws in a partial plan as its heuristic. This heuristic is not admissible nor informative, because some flaws can be solved by adding a single causal link while others cannot be refined due to existing constraints, which means that the partial plan

is a dead end. Gerevini and Schubert [20] showed that counting the number of open conditions often gives better results, but this method suffers from the same weaknesses. Lack of heuristic guidance has been seen as one of the fundamental weaknesses of partial-order planning. With the advent of planning systems based on planning graphs (e.g. GraphPlan [3] and FF [32]), a resurgence in partial-order planning was sparked by planning systems such as RePOP [42] and VHPOP [57], which tried to incorporate these new heuristics in partial-order planners and showed that partial-order planning also benefit from these new heuristics.

The heuristic used by RePOP ignores all the threats and only considers open conditions. Given a partial plan $\xi = \langle A, L, O, B \rangle$ and all the open conditions F_{oc}, RePOP constructs a *serial graph* until it reaches a fact layer where all open conditions F_{oc} are true. Let $lev(l)$ be the index of the level where the literal l first appears in a fact layer. Let $l_s = \max_{l_i \in F_{oc}} lev(l_i)$. We can achieve l_s by adding an action $a \in A$ to the plan that achieves l_s. The set of open conditions is than updated as $F'_{oc} = F_{oc} \cup a_{precs} \setminus a_{effects}$. We can express the costs of ξ as:

$$cost(F_{oc}) = a + cost(F_{oc} \cup a_{precs} \setminus a_{effects}) \tag{2.14}$$

The paper on RePOP [42] does not specify how an action a is chosen. It does not follow FF's method of calculating the difficulty of achieving an action or preferring actions that appear earlier in the fact layer. This is, however, a step up from UCPOP's method of counting the number of flaws and takes positive interaction of actions into account.

VHPOP entered the IPC-3[39] competition and was awarded 'best newcomer'. Its heuristic is based on h^{add} used by HSP [4]. Given a literal l, let $a_l \subseteq A$ be all the actions that have an effect which can be unified with l. The cost of achieving l can then be defined as:

$$cost(l) = \begin{cases} 0 & \text{if } l \in s_0 \\ \min_{a \in a_l} cost(a) & \text{if } a_l \neq \emptyset \\ \infty & \text{otherwise} \end{cases} \tag{2.15}$$

The cost of achieving an action a is then defined as:

$$cost(a) = 1 + \Sigma_{p \in a_{precs}} cost(p) \tag{2.16}$$

The heuristic value of a partial plan $\xi = \langle A, L, O, B \rangle$ and all the open conditions F_{oc} is defined as:

$$h^{add}(\xi) = \Sigma_{\xrightarrow{l} a_i} \in F_{oc} cost(l) \tag{2.17}$$

Finally, to take positive interactions into account, the h^{add} heuristic is altered as follows:

$$h_r^{add}(\xi) = \Sigma_{\xrightarrow{l} a_i \in F_{oc}} \begin{cases} 0 & \text{if } \exists a_j \in A \text{ such that an} \\ & \text{effect of } a_j \text{ can be unified} \\ & \text{with } l \text{ and } a_i \prec a_j \notin O \\ cost(l) & \text{otherwise} \end{cases} \tag{2.18}$$

If two partial plans have the same heuristic value VHPOP uses the *estimated remaining effort* as a tie-breaker. This is equal to the number of causal links that need to be introduced in the partial plan before all flaws are resolved. The difference with the h_r^{add} heuristic is that literals that can be achieved by adding a causal link to the initial fact cost 1 instead of 0.

Both RePOP and VHPOP take positive interactions into account, but the difference is that VHPOP tries to look for the minimal amount of actions per subgoal and assumes subgoal independence, whereas RePOP chooses actions arbitrarily and relies on the constructed planning graph to enforce the action orderings. Furthermore VHPOP actively tries to reuse actions whenever it can whereas in RePOP this appears to be more of an happy accident when it chooses the same action from the planning graph. Comparing the informativeness of both heuristics, it is clear that the one used by VHPOP is more informative on most planning problems.

2.5.2 Flaw selection strategies

Once a partial plan has been selected to be refined, we must choose which flaw we want to resolve. It has been shown in the literature that the order in which flaws are resolved can have a considerable effect on how quickly planning problems are solved [48]. A flaw is either an *open condition*(oc), *separable threat*(s), or *non-separable threat*(n). A policy might prefer to solve open conditions before threats, or the other way around. In case we find a partial plan with many flaws of the same type a tie-breaking algorithm is applied to determine which flaw to solve. In this section we follow the notation introduced by Pollack, et al. [48], given as: $\{flawtype\}_{repaircostrange} tie - breakingstrategy$.

The flaw types have been described above, the *repair cost range* means the number of ways a flaw can be resolved and the *tie-breaking strategy* is there in case there is more than one flaw which meet the first two criteria.

UCPOP, for example, tries to repair threats before open conditions. It uses the time at which each flaw was introduced as a tie-breaking mechanism: LIFO. The last flaw that was introduced is the first to be resolved. We can notate this strategy as: $\{n, s\}LIFO \setminus \{ocLIFO\}$. By choosing LIFO as a tie-breaker the planner focuses on the preconditions of the last introduced action and only considers the next precondition of that action when all flaws introduced by the former are resolved.

However, further investigation found that delaying separable threats dominates this strategy. This strategy has been dubbed DSep and is given as $\{n\}LIFO \setminus \{ocLIFO\} \setminus \{sLIFO\}$. The rationale behind delaying resolving separable threats is that these might be resolved due to reusing actions to resolve open conditions. Non-separable threats are unlikely to be resolved by solving open conditions or separable threats, moreover it might be the case that there is no way to solve non-separable threats, which means the plan is a dead end. This follows the principle used in solving CSP problems where

working on the most constrained sets leads to earlier dead-end detection and by branching on the most constraint set first might reduce some sets to unit clauses and reduce the search space. The *Least-Cost Flaw Repair*(LR) strategy follows the same idea and is defined as: $\{oc, n, s\}_i \mid i \in 0, 1, \ldots, \infty$. This idea has been shown to be very effective in reducing the search space. Further experimentations developed a strategy dubbed $ZLIFO$ which is defined as: $\{n\}LIFO \setminus \{oc_0\}LIFO \setminus \{oc_1\}New \setminus \{oc\} \setminus \{s\}LIFO$. It prefers resolving non-separable threats over open conditions flaws that are separated into different categories. It prefers to solve open conditions that can be resolved with *zero-commitment*, which means the planner has no choice in how to resolve this open condition and does not commit itself beyond what it *must* do. Next it considers open conditions that can only be resolved by introducing a *new* action to the partial plan, and lastly it considers all other open conditions before resolving separable threats. This strategy, $ZLIFO$, produces smaller search spaces compared to the *LR* strategy on most planning problems. However, if the LR strategy is altered to: $\{oc, n\}_i \mid i \in 0, 1, \ldots, \infty \setminus \{s\}_i \mid i \in 0, 1, \ldots, \infty$ it performs at least as well and sometimes better than $ZLIFO$. It seems that delaying separable threads pays off as a strategy, but apart from that no hard results are found which work consistently on all domains.

VHPOP, for that very reason, decides to use a set of flaw selection strategies and runs four planners in parallel, each with a different flaw selection strategy. VHPOP introduces new flaw types related to open conditions to define their flaw strategies. These are:

- t static open condition; if the open condition is a static literal (i.e. it cannot be achieved by any action), then is must be achieved by the initial action.
- l local open condition; using the $\{oc\}LIFO$ strategy the planner focuses on the last introduced precondition. Local open preconditions are all preconditions of the last introduced action.
- u unsafe open condition; an open condition is unsafe if a causal link to that open condition would be threatened. Prioritising these open conditions before others might lead to quicker dead-end detection, thus pruning the search space.

As a tie-breaking heuristic VHPOP uses the estimated remaining effort, which is also used to select which partial plan to work on. This tie-breaking heuristic is denoted as MW_{add}. The flaw selection strategies used by VHPOP then are:

- MW-Loc: $\{n, s\}LR \setminus \{l\}MWadd$
- MW-Loc-Conf: $\{n, s\}LR \setminus \{u\}MWadd \setminus \{l\}MWadd$
- LCFR-Loc: $\{n, s, l\}LR$
- LCFR-Loc-Conf: $\{n, s, u\}LR \setminus \{l\}LR$

Note that VHPOP, like RePOP, only works with grounded actions so the set of separable threats is always empty.

2.6 Domain analysis

State-of-the-art planners rely heavily on heuristics to guide a planner towards the goal. However, heuristics alone are not sufficient to guarantee good performance even if they are *near perfect*. How informative is a heuristic h? One method of gauging this is to compare it with the optimal heuristic h^* on a set of problems and measuring the difference between the values returned by both heuristics. One could say the closer the heuristic estimates of h are to h^* the more informative it is. Unfortunately, the computational effort of calculating h^* exactly is as hard as solving the original planning problem. A search would no longer be required, because given any state we select the successor whose heuristic estimate is one less until we reach the goal.

Other techniques have been developed to overcome the limitations of heuristics. Some techniques deal with how to deal with *plateaus*, which are part of the search space where all successors of a state s have the same or a worse heuristic value than s. In these cases the heuristic offers no guidance to a better state. Some planners – e.g. Fast Downward – use multiple heuristic estimators while others – e.g. Fast Forward – use a systematic search strategy in order to escape plateaus. While these techniques have proved to be useful if the number of successor states per state is very large they will not help to speed up the search. Instead some planners use pruning techniques to only consider a subset of all the successor states to explore. These pruning techniques have proved to be very effective in larger problem instances.

In this section we will describe techniques relevant to our work that perform *domain analysis*. Performing domain analysis means that we try to extract information from a planning problem prior to search, which can be used to either prune or guide the search. For example, domain analysis can detect certain structures or sub-problems in a planning problem that can be exploited or solved by a specialised planner. For example, STAN [17] searches for sub-problems in a planning task, for example path finding problems, and uses specialised solvers to solve those. RealPlan [58] on the other hand found that many planners find it harder to solve problems when more *resources* are available. To solve this paradox RealPLan decouples resource allocation from the planning task in such a way that adding more resources makes the planning problem easier.

Other techniques, use *symmetry breaking* to simplify the problem. If multiple objects have the same initial properties and need to achieve the same goals then we can reduce the search space that needs to be considered by only considering one of them.

In this section we will touch upon those domain analysis techniques that are relevant to our work. Most notable are symmetry breaking and the transition from a typed planning task to a typed multi-valued planning task. We will first present the translation technique we use called TIM. Next we present the transition used by Fast Downward and MIPS [9]. Finally we expand on some work on *(near) symmetry* breaking.

2.6.1 TIM analysis

TIM [15], short for *Type Inference Module*, analyses a planning task ($\Pi = \langle T, O, P, A, s_0, s_g \rangle$, where $T = \emptyset$) and infers a type structure of the domain and extracts state invariants. Since we are given a *typed* planning task we will focus on the methods used by

TIM to extract state invariants. We will only touch upon the type inference techniques used by TIM where it is necessary to explain how state invariants are extracted.

Definition 29 — Property

A property is a predicate subscripted by a number between 1 and the arity of that predicate. Every predicate of arity n defines n properties.

For example, a predicate $(at\ X\ Y)$ yields the properties at_1 and at_2. Operators in a planning problem can either remove or add properties. TIM encodes these relationships between properties using *transition rules*. Given that a typed planning task is a tuple $\Pi = \langle T, O, P, A, s_0, s_g \rangle$, TIM builds a transition rule for each variable of each operator.

Definition 30 — Transition Rule

A transition rule is an expression of the form: $property \overset{op}{\Rightarrow} property \rightarrow property$ in which the three components are bags of zero or more properties called enablers, start and finish, respectively. If a bag is empty we denote it by $null$.

We denote the following operators for bags: $\ominus$ to denote difference, $\otimes$ to denote intersection, $\oplus$ to denote union and $\sqsubseteq$ to denote inclusion. We will use square brackets to denote members of a bag.

Given an operator $op = \langle name, parameters, precs, effects \rangle \in \Pi_A$ and a parameter $p_i \in parameters$, TIM constructs a triplet of bags of properties termed a *property relating structure* (PRS). The first bag consists of all the properties of the preconditions $\langle p, V \rangle \in precs \mid V = p_i$. For example, given the precondition $at\ X\ Y$ and the parameter being considered is X than at_0 will be added to this bag. This bag, called *enablers*, contains the enablers that will be used in the formation of the transition rules. The second bag of properties, called *deleted preconditions*, is formed from all of the preconditions $\langle p, V \rangle \in precs \mid \exists e \in effects^{-} \langle p, V \rangle = e \wedge V = p_i$. The thrid bag of properties, called *added effects*, is formed from all the effects $\langle p, V \rangle \in effects^{+} \mid p = p_i$.

From this structure the transition rules are constructed according to the following formula:

$$enablers \ominus deleted\ preconditions \overset{op}{\Rightarrow} deleted\ preconditions \rightarrow added\ effects \tag{2.19}$$

Example 2.6.1 *Consider the following operators from the* Zeno *domain:*

The Fly *operator:*

- name = *Fly*
- parameters = $\{ a = \langle aircraft, \emptyset \rangle, c1 = \langle city, \emptyset \rangle, c2 = \langle city, \emptyset \rangle, fl1 = \langle flevel, \emptyset \rangle, f2 = \langle flevel, \emptyset \rangle \}$
- precs = $(at\ a\ c1) \wedge (fuel - level\ a\ fl1) \wedge (next\ fl2\ fl1)$
- effects = $\neg(at\ a\ c1) \wedge (at\ a\ c2) \wedge \neg(fuel - level\ a\ fl1) \wedge (fuel - level\ a\ fl2)$

The Board *operator:*

- name = *Board*

- parameters = $\{p = \langle person, a' = \langle aircraft, \emptyset\rangle, c = \langle city, \emptyset\rangle\}$
- precs = $(at\ a'\ c) \wedge (at\ p\ c)$
- effects = $\neg(at\ p\ c) \wedge (in\ p\ a')$

And the Debark *operator:*

- name = *Debark*
- parameters = $\{p' = \langle person, a'' = \langle aircraft, \emptyset\rangle, c' = \langle city, \emptyset\rangle\}$
- precs = $(at\ a''\ c') \wedge (in\ p\ a'')$
- effects = $\neg(in\ p'\ a'') \wedge (at\ p\ c')$

By considering the parameter $a = \langle aircraft, \emptyset\rangle$*, the following PRS will be built:*

$$enablers = at_1, fuel - level_1 \tag{2.20}$$
$$deleted\ preconditions = at_1, fuel - level_1 \tag{2.21}$$
$$added\ effects = at_1, fuel - level_1 \tag{2.22}$$

By considering the parameter $c1 = \langle city, \emptyset\rangle$ *we obtain:*

$$enablers = at_2 \tag{2.23}$$
$$deleted\ preconditions = at_2 \tag{2.24}$$
$$added\ effects = \tag{2.25}$$
$$\tag{2.26}$$

By considering the parameter $c2 = \langle city, \emptyset\rangle$ *we obtain:*

$$enablers = \tag{2.27}$$
$$deleted\ preconditions = \tag{2.28}$$
$$added\ effects = at_2 \tag{2.29}$$
$$\tag{2.30}$$

By considering the parameter $fl1 = \langle flevel, \emptyset\rangle$ *we obtain:*

$$enablers = fuel - level_2, next_2 \tag{2.31}$$
$$deleted\ preconditions = fuel - level_2 \tag{2.32}$$
$$added\ effects = \tag{2.33}$$
$$\tag{2.34}$$

By considering the parameter $fl2 = \langle flevel, \emptyset\rangle$ *we obtain:*

$$enablers = next_1 \tag{2.35}$$
$$deleted\ preconditions = \tag{2.36}$$
$$added\ effects = fuel - level_2 \tag{2.37}$$
$$\tag{2.38}$$

By considering the parameter $p = \langle person, \emptyset \rangle$, the following PRS will be built:

$$enablers = at_1 \tag{2.39}$$
$$deleted\ preconditions = at_1 \tag{2.40}$$
$$added\ effects = in_1 \tag{2.41}$$

By considering the parameter $a' = \langle aircraft, \emptyset \rangle$, the following PRS will be built:

$$enablers = at_1 \tag{2.42}$$
$$deleted\ preconditions = \tag{2.43}$$
$$added\ effects = in_2 \tag{2.44}$$

By considering the parameter $c = \langle city, \emptyset \rangle$ we obtain:

$$enablers = at_2, at_2 \tag{2.45}$$
$$deleted\ preconditions = at_2 \tag{2.46}$$
$$added\ effects = \tag{2.47}$$
$$\tag{2.48}$$

By considering the parameter $p' = \langle person, \emptyset \rangle$, the following PRS will be built:

$$enablers = in_1 \tag{2.49}$$
$$deleted\ preconditions = in_1 \tag{2.50}$$
$$added\ effects = at_1 \tag{2.51}$$

By considering the parameter $a'' = \langle aircraft, \emptyset \rangle$, the following PRS will be built:

$$enablers = at_1 \tag{2.52}$$
$$deleted\ preconditions = in_2 \tag{2.53}$$
$$added\ effects = \tag{2.54}$$

By considering the parameter $c' = \langle city, \emptyset \rangle$ we obtain:

$$enablers = at_2 \tag{2.55}$$
$$deleted\ preconditions = \tag{2.56}$$
$$added\ effects = at_2 \tag{2.57}$$
$$\tag{2.58}$$

A potential problem with this encoding is the PRS for the parameter $a = \langle aircraft, \emptyset \rangle$. The PRS *exchanges* the properties at_0 and $fuel - level_1$. It might be possible that the domain contains objects that do not require a fuel level to fly (although that is not the case here), in which case we cannot distinguish them from objects that do require fuel to fly. To solve this problem TIM splits the PRS as follows:

$$enablers = at_1, fuel - level_1 \tag{2.59}$$
$$deleted\ preconditions = at_1 \tag{2.60}$$
$$added\ effects = at_1 \tag{2.61}$$

Here fuel-level is no longer part of the exchanged properties but is an enabler for the plane to fly. Likewise:

$$enablers = at_1, fuel - level_1 \tag{2.62}$$
$$deleted\ preconditions = fuel - level_1 \tag{2.63}$$
$$added\ effects = fuel - level_1 \tag{2.64}$$

The plane being at a location is now an enabler for it to change its fuel level. In general a PRS will be split when multiple properties appear both in the *deleted preconditions* and *added effects* bags. Eventually we end up with the following transition rules:

$$fuel - level_1 \overset{fly}{\Rightarrow} at_1 \rightarrow at_1 \tag{2.65}$$
$$at_1 \overset{fly}{\Rightarrow} fuel - level_1 \rightarrow fuel - level_1 \tag{2.66}$$

Note that both transition rules delete a property but gain another property in return. This is not always the case. Consider for example the transition rule for the parameter $fl1 = \langle flevel, \emptyset \rangle$:

$$null \overset{fly}{\Rightarrow} fuel - level_2 \rightarrow null$$

This is an example of a *decreasing attribute transition rule* because a property is removed but no property is added. The transition rule for the parameter $fl2 = \langle flevel, \emptyset \rangle$:

$$null \overset{fly}{\Rightarrow} null \rightarrow fuel - level_2$$

is an example of an *increasing attribute transition rule* because a property is gained without having to delete one. These transition rules are used to construct *property spaces* and *attribute spaces*.

Definition 31 — Property space
A property space is a tuple $\langle P, T, S, C \rangle$ where:

- P is a set of properties.
- T is a set of transition rules.
- S is a set of states, where a state is a bag of properties.
- C is a set of domain constraints.

Definition 32 — Attribute space
A property space is a tuple $\langle P, T, C \rangle$ where:

- P is a set of properties.
- T is a set of transition rules.
- C is a set of domain constraints.

To construct the property and attribute spaces, TIM groups together transitions rules whose properties they affect overlap. So given a transition rule $enablers \stackrel{op}{\Rightarrow} deleted\ preconditions \rightarrow added\ effects$ we unify $deleted\ preconditions$ and $added\ effects$ to *seed* a property or attribute space. Given two transition rules whose start or finish bags overlap we unify both their start and finish bags and use the result to seed a property or attribute space. Next we associate each transition rule with one of the created collections of properties. This is done by identifying which collection a property in the start of finish bag of a transition rule belongs to. Note that there can never be any ambiguity because every property is part of a single set. Those collections which do not contain an attribute transition rule are made into *property spaces* while those that do are made into *attribute spaces* .

Example 2.6.2 *Continuing our example above, the following collections of properties are formed:*

- at_1, in_1
- $fuel - level_1$
- $fuel - level_2$
- at_2

The transition rules are associated with each collection as follows:

$[at_1, in_1]$	$fuel - level_1 \overset{fly}{\Rightarrow} at_1 \rightarrow at_1$
	$at_1 \overset{board}{\Rightarrow} at_1 \rightarrow in_1$
	$at_1 \overset{debark}{\Rightarrow} in_1 \rightarrow at_1$
$[in_2]$	$at_1 \overset{board}{\Rightarrow} null \rightarrow in_2$
	$at_1 \overset{debark}{\Rightarrow} in_2 \rightarrow null$
$[at_2]$	$at_2 \overset{fly}{\Rightarrow} at_2 \rightarrow null$
	$null \overset{fly}{\Rightarrow} null \rightarrow at_2$
	$at_2 \overset{board}{\Rightarrow} at_2 \rightarrow null$
	$at_2 \overset{debark}{\Rightarrow} null \rightarrow at_2$
$[fuel - level_1]$	$at_1 \overset{fly}{\Rightarrow} fuel - level_1 \rightarrow fuel - level_1$
$[fuel - level_2]$	$next_1 \overset{fly}{\Rightarrow} null \rightarrow fuel - level_2$,
	$next_2 \overset{fly}{\Rightarrow} fuel - level_2 \rightarrow null$

Of these sets of properties only at_1, in_1 and $fuel - level_1$ contain no attribute transition rules and will be converted into property spaces. The other sets of properties will be converted into attribute spaces. Next TIM checks which objects inhabit which spaces. For every object o we construct a bag of O_i properties that are true in the initial state. For each property state $\langle P, T, S, C\rangle$ we check if $P \otimes O_i \neq \emptyset$; if that is the case than o is added to the state S. Next TIM expands the state by applying the transition rules on the properties in the state and adds the results to the state too. Consider the property space constructed for the properties at_1, in_1; objects can *exchange* properties by applying the operators. Attribute spaces, by contrast, can either obtain new properties or lose them. For every attribute space $\langle P', T', C'\rangle$ TIM checks for objects where the increase rules apply and we add those to C'.

Definition 33 — Balanced Properties
Properties which are part of a property state are *balanced*.

Definition 34 — Unbalanced Properties
Properties which are part of an attribute space either decrease or increase in number and are *unbalanced*.

Example 2.6.3 *Given a* Zeno *planning problem where the initial state consists of a* Person p *and an* Airplane a *at a location, where the airplane has some* Fuel Level f*, the property space for the properties* at_1*,* in_1 *would be initiated as:*

- $P = at_1, in_1$
- $T = fuel - level_1 \overset{fly}{\Rightarrow} at_1 \rightarrow at_1, at_1 \overset{board}{\Rightarrow} at_1 \rightarrow in_1, at_1 \overset{debark}{\Rightarrow} in_1 \rightarrow at_1$
- $S = [at_1], [at_1]$
- $C = p, a$

We can apply the transition rule $at_1 \overset{board}{\Rightarrow} at_1 \rightarrow in_1$ *to the state* S *which will become* $S = [at_1], [at_1], [in_1]$.

Extract State Invariants

Based on the constructed property states, TIM assigns to each object a *type*. A type can be represented by the set of property and attribute spaces an object is part of. Consider two objects o and o' with their types T_o and T'_o, respectively. We say o and o' are of the same type if $T_o = T'_o$. If $T_o \subset T'_o$ than o's type is a super type of o''s.

TIM extracts *identity invariants*, *state membership invariants*, and *uniqueness invariants* from the property spaces. In addition TIM extracts *fixed resource invariants* from the operator schemas and the initial state of a planning problem.

- *Identity invariants* constrain the number of times a property can occur in a domain. In the Zeno example, if every plane is at a location in the initial state we know that they will *always* have that property at_1 because they are not part of an attribute space that deletes this property or part of a space which exchanges at_1 for another property. So the following is true for objects of type airplanes:

  $$\forall_{o \in \Pi_O \mid Type(o)=airplane} \forall_{x \in \Pi_O \mid Type(x)=city} \forall_{y \in \Pi_O \mid Type(y)=city} (at\ o\ x) \wedge (at\ o\ y) \implies x = y$$

  This form is generalised by TIM so that instead of one occurrence of a property in a state it can have multiple occurrences in a state.

- *State membership invariants*, constructed for each property state $\langle P, T, S, C \rangle$, check if all the properties p_o which hold for an object o in a state are a subset of $s \in S$. If this is the case than o is an invariant. For example, given a person p in the Zeno domain at a location we check the following sentence:

  $$\exists_{c \in \Pi_O \mid Type(c)=city} ((at\ o,\ c))$$

  If a state contains multiple properties we need to check membership of each one of them.

- *Uniqueness invariants* constrain objects from being part of more than two states of a single property space. For example, the following sentence is true if a person is not in a plane and at a city at the same time:

  $$\forall_{p \in \Pi_O \mid Type(p)=person} \neg(\exists_{c \in \Pi_O \mid Type(c)=city} (at\ p\ c)) \wedge \exists_{a \in \Pi_O \mid Type(a)=airplane} (in\ p\ a))$$

  TIM generalises this sentence to make sure that an object is only part of a single state of a property state.

The last invariant TIM extracts are *fixed resource invariants*, which do not depend on the property spaces but rather on the structure of the actions in a planning task and its initial state. Fixed resource invariants hold over predicates instead of objects. For example, if the Zeno domain was encoded in such a way that the location of a plane would be encoded as $(at\_plane\ x)$ then the following transition rules would be constructed:

- $null \stackrel{fly}{\Rightarrow} at\_plane\ x \rightarrow null$
- $null \stackrel{fly}{\Rightarrow} null \rightarrow at\_plane\ x$

These are both attribute spaces so are not considered by the other three invariant extraction methods. However, it is clear that whenever the predicate $(at\_plane\ x)$ is removed a new one is added to a state. In other words, the predicate is balanced. So if in the initial state only a single $(at\_plane\ x)$ predicate is true than we can extract the following invariant:

$$| \{x : (at\_plane\ x)\} | = 1$$

To infer these invariants TIM evaluates all actions and checks if a property is equally exchanged in all actions. If this is the case than the above invariant can be extracted. This means that there can never be more invariable predicates than there exists in the initial state. Sometimes a less strict invariant must be constructed. Consider a domain where there are two $(at\_plane\ x)$ predicates true in the initial state. If the fly action is executed so that the variables of one of the predicates becomes equivalent to the other then only a single predicate will be in any state reachable from there. So in those cases the weaker invariant

$$| \{x : (at\_plane\ x)\} | \leq k$$

is extracted, where k is the number of instances of that predicate in the initial state.

In order to extract more invariants TIM evaluates all the constructed property and attribute spaces for each *type* of object and applies the same techniques as described above to extract more invariants. For example – while no new invariants are extracted in this case – the property state associated with the state $[at_1, in_1]$ will be reevaluated for objects of the type $airplane$ and $person$. Also, TIM is able to correct cases where a property is mistaken to be part of a property space and splits the offending property into an attribute space.

2.6.2 Fast Downward / MIPS analysis

As we have discussed in Section 2.4.2 Fast Downward uses a translation technique to translate a typed planning task into a typed multi-valued planning task. In this section we will discuss the technique Fast Downward uses.

To simplify the discussion we borrow some definitions from TIM. The original aim for this technique is to minimise the space required to encode a state. Unlike TIM, this method searches for properties per *object* instead of *type*. Given a typed planning task $\Pi = \langle T, O, P, A, s_0, s_g \rangle$, we define a mapping $predicate_{(i,o)} : \{grounded\ atoms\} \rightarrow \mathbb{N}$ such that $predicate \in P$, $i \in \{0, \ldots, arity\ of\ predicate\}$, and $o \in O$. Given a set of atoms S, $p_{(i,o)}$ is equal to the number of facts $\langle p', V \rangle \in S$ such that $p = p'$ and $D = \{o\}$, where $\langle t, D \rangle \in V_i$.

For any object $o \in O$, predicate $p \in P$, and $i \in \mathbb{N}$, for which $p_{(i,o)} > 1$ we use the standard PDDL encoding. If $p_{(i,o)} = 1$ and there is no grounded action $a \in A$ that has

an effect $\langle p', V \rangle \in a^{-}_{effects}$ such that $p' = p$ and $D \in V_i = \{o\}$ then $p_{(i,o)}$ is balanced. If $p_{(i,o)} = 0$ and no action can affect it then we can ignore it.

In most cases, however, most atoms are affected by actions. However, if we can find a set of predicates and indexes for an object o such that the sum of these mappings for any sequence of actions are balanced then we can encode this set as a multi-valued variable. For example, given a driverlog domain a truck t and the location objects $s1$, $s2$, and $s3$ then $at_2(t)$ is balanced *iff* Π_{s_0} contains exactly one atom from the following set: { *(at t s1)*, *(at t s2)*, *(at t s3)* }. However, not all properties are so well behaved. For example, the property $at_2(d)$ where d is of type *driver* is not balanced due to the *board* operator which removes $at_2(d)$ but adds $in_2(d)$. In order to capture these cases, Fast Downward searches for all possible combinations of properties which are *balanced*.

The final step is to prune the set of values that can be assigned to a state variable. For example, it might be the case that the road network of a driverlog domain is disconnected. In this case we wish to prune any values from the variable domain which are unreachable. In order to prune these domains a reachability analysis is conducted; while the method used is different the results are the same as when constructing an RPG until the level-off point. We prune any atoms from the domains of the found state variables that do not appear in the final fact layer.

2.6.3 Symmetry breaking

The aim of our thesis is to develop ideas for working on and solving large planning problems. In the international planning competition of 1998 a planning domain called *Gripper* was introduced whose domain contains a robot with two grippers that can pick up and drop balls. The robot can move back and forth between two rooms and the goal is to move all the balls that are initially in one of the rooms to the other room. Due to the number of balls (a few hundred) the planner is confronted with many possible actions that are applicable in any given state. A planner could get stuck if it tried to consider all possible orderings in which the balls could be picked up and by which gripper they should be picked up. The search space of this problem scales exponentially based on the number of balls. It is easy to see how, if we disregard the robot and grippers, the number of states that we can describe by distributing the balls over the two rooms is 2^n, where n is the number of balls. The maximum number of actions that can be applied in any state is equal to $1 + 2n$ – this is the situation where the robot is in the room where all the n balls are. The operators are then to move the robot to the other room (1 action) or to pick up a ball with the left gripper (n possible actions) or pick up a ball with the right gripper (n possible actions).

If the goal of this problem is to get all the n balls from one room to the other then it is clear to us humans that it does not matter: 1) Which ball to pick up first; and 2) Which gripper to use to pick up the ball. This is a method called *symmetry breaking* and vastly reduces the search space. In this thesis we want to solve bigger problem instances so being able to detect symmetry relations and exploiting them to reduce the search space is vital. In this section we present previous methods that have been developed to detect and exploit symmetry relationships.

Definition 35 — Symmetry Group

Given a transition graph $\langle S, L, A, s_0, S_g \rangle$ (see Definition 22) where $L = \emptyset$ and a property $E : S \to X$ (for some set X) then a *symmetry group* is a set of pairs $\langle \alpha, \beta \rangle$ such that:

- $\alpha : S \to S$ is a permutation on S,
- $\beta : A \to A$ is a permutation on A,
- for every pair of states $s_1 \in S$ and $s_2 \in S$ and a transition $a \in A$, $(\alpha(s_1), \alpha(s_2)) \in \beta(t)$ *iff* $(s_1, s_2) \in t$, and
- $E(s) = E(\alpha(s))$ for all $s \in S$.

In other words a symmetry group can be created by finding an automorphism of the transition graph that maintains its structure. Unfortunately, finding all the possible permutations of a transition graph is very expensive. In fact it is harder than solving the graph isomorphism problem [40]. Most methods used to find *symmetry groups* in the context of planning problems constrain the permutations that are considered. For example, given a *symmetry group* $G = \{\langle \alpha, \beta \rangle\}$ we might set a condition that a subset of the states $S' \subseteq S$ should not be changed, i.e. for every $s' \in S'$ it must hold that $\alpha(s') = s'$: this is the function of E.

To see how this restriction is useful, consider a symmetry group G, a transition graph $\langle S, L, A, s_0, S_g \rangle$ and assume we have found a sequence of actions $\langle a_0, a_1, \ldots, a_n \rangle$. We could then apply a permutation $\langle \alpha, \beta \rangle \in G$ on the transitions such that we find the sequence of actions $U = \langle \beta(a_0), \beta(a_1), \ldots, \beta(a_n) \rangle$ which is a solution to the transition graph $\langle \alpha(S), L, \beta(A), \alpha(s_0), \alpha(S_g) \rangle$. However, there is no guarantee that $\alpha(s_0) = s_0$, so the found solution U is not a solution to the actual problem. If we, however, restrict the permutation we allow by enforcing the rule that for any permutation $\langle \alpha, \beta \rangle \in G$ $\alpha(s_0) = s_0$ and $\alpha(s_g) = s_g$, then the solution U is also a solution to the original problem.

Symmetrical objects and transitions

Methods to detect symmetry have initially been developed in the context of model checking [10] and later been extended to break symmetry in constraint satisfaction problems [52, 61] mostly to reduce the search space to a more manageable size. These methods have been adopted and integrated in planning systems to make planning systems more scalable in large domains that exhibit a lot of symmetry, such as the *Gripper* domain explained above. One of the first papers to explore the use of symmetry in planning systems [16] searches for groups of symmetrical objects, defined in the paper as objects that are indistinguishable from one another in terms of their initial and final configurations. So in the case of our *Gripper* example above all the balls and the grippers are put into two symmetric groups. These symmetric groups are used to detect symmetric groups of actions. Given two grounded actions, if every pair of objects – one for each parameter of each action – are in the same symmetric group, then both actions are part of the same symmetric group.

These symmetric groups are next used in the planner STAN [17], based on the GraphPlan [3] architecture for pruning search trees. A shortcoming of this method – that was solved in a subsequent paper [18] – is that the symmetric groups found for the initial fact were not updated, which means that symmetries that occurred later in a plan were not detected.

Subsequent work focused on breaking symmetry in the context of forward-chaining planners. This work focused on exploiting symmetry groups by : (1) proving that any two states s and s' are *symmetrical* [46]; and (2) given a state s and two transitions t and t', if both transitions are *symmetrical* then we only have to consider one of the transitions and we can ignore the other [55].

The method most relevant to our work is the latter. This work is very similar to the techniques in STAN [17]: the definitions given are more generally applicable but the underlying idea and methods used are identical. Using Definition 35 and given a state s we search for an automorphism that produces a set of symmetry groups G that maps s to itself, i.e. given any symmetry group $\langle \alpha, \beta \rangle \in G$ the following holds: $\alpha(s) = s$. Then any pair of transitions t and t' that are applicable to s are *interchangeable iff* $\beta(t) = t'$.

Theorem 2.6.1 *(adapted from [55]) Let $\langle a_0, a_1, \ldots, a_n \rangle$ be an action sequence that is applicable to s_0 which visits the sequence of states $\langle s_0, s_1, \ldots, s_n \rangle$. Let a'_i be interchangeable with a_i in state s_i. Then there is a sequence of actions $\langle a'_{i+1}, \ldots, a'_n \rangle$ that visits the sequence of states $\langle s'_{i+1}, \ldots, s'_n \rangle$ such that $E(s_j) = E(s'_j)$ for every $j \in \{i, \ldots, n\}$.*

Proof: Because a_i and a'_i are interchangeable, there is $\langle \alpha, \beta \rangle \in G$ such that $\alpha(s_i) = s_i$ and $\beta(a_i) = a'_i$. Define $\langle s'_i, \ldots s'_n \rangle$ by $s'_j = \alpha(s_j)$ and $\langle t'_i, \ldots, t'_n \rangle$ by $t'(j) = \beta(t_j)$ for all $j \in \{i, \ldots, n\}$. By definition of symmetry groups $(E_{s'_j}) = E(\alpha(s_j)) = E(s_j)$ for all $j \in \{i, \ldots, n\}$.

This leaves open the problem of actually finding an automorphism for a planning problem. It is a very hard problem to find all of the possible automorphism; however, some automorphisms are easy to find., e.g. finding groups of symmetrical objects. In [55] these symmetrical groups are constructed by grouping any objects o and o' if (1) there are no actions that contain *static* preconditions which includes one of the objects but not the other and (2) for any goal fact where o is part of there must be another goal that is equivalent but swaps o for o'. This kind of symmetry is called *functional symmetry* because there is no functional difference between o and o', previously defined in [16]. These function symmetry groups can then be used to find symmetry groups of actions described above.

Symmetrical states

Instead of searching for objects and transitions that are part of the same symmetry group, another branch of research searches for *states* that are symmetrical [46]. This technique can be used by a forward-chaining planning as follows. Whenever we want to expand a state s we can check if a state s' has already been extended that is symmetrical to s. If that is the case then we can ignore s. To prevent having to check

if s is symmetrical with any previously expanded state s' we store in canonical form all states it has encountered before, because finding the *lex-minimal* state is a NP-hard problem [40] and so this state is approximated. This line of research is not relevant to our work so we refer to [46] for more details.

Almost symmetry

An observation made by [49] is that in many planning problems a pair of objects are not part of a symmetric group, but the sequence of actions that needs to be executed to get both objects to their goal state is equivalent. So the facts that are true for both objects are *almost* identical. For example, consider a *Depots* domain where two crates are at the same place but stacked on top of different pallet, and both need to be stacked on different pallets at the same location. While these crates are not symmetrical, both need to be unstacked, moved to the other place and finally stacked on their respective pallets to reach the goal. The methods discussed so far will not detect this form of symmetry.

Definition 36 — Almost symmetry

Given a typed planning task $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ and two objects $o \in O$ and $o' \in O'$. Then we define two sets of *properties* P_o and P'_o such that each set contains all the properties that correspond to the facts in s_0 and s_g for o and o', respectively. Then o and o' are almost symmetrical iff they have the same type and P_o and P'_o contain the same set of properties.

We can construct the set of almost symmetrical objects by constructing a coloured graph G where all the $o \in O$ objects in a typed planning task are added as a vertex, with a unique colour. Next, for every object $o \in O$ we collect all the facts from the initial state $S \subseteq s_0$, such that every fact $\langle p, V \rangle \in S$ contains a variable domain $\langle t, D_v \rangle \in V \mid o \in D_v$. Next we add S as a vertex to G; the colour is determined by the name, arity and the set of types of the predicate p and the set of indexes of o in each of the facts $s \in S$. The same is done for the goal facts. Next we find symmetries of this graph based on the colourings of the nodes ([49] uses a tool called NAUTY [41]). The found symmetrical groups are those objects which are *almost symmetrical*.

The way almost symmetrical groups are found is by *abstracting* the planning problem. This is accomplished by – given an object o – ignoring all the identities of all the other objects in an atom of the initial or goal node whose identity is not the same as o and substituting these with the *type* of this object. If an object o has similar relations with other objects of similar types in the initial and goal state compared to another object o' then these objects are *almost symmetrical*.

Example 2.6.4 *Consider the* Blocksworld *domain depicted in Figure 2.3. By using the algorithm outlined above we end up with the graph depicted in Figure 2.4.*

From this graph we can easily see that the nodes { b1, b3, b5 } *are interchangeable, and the same is true for the set of nodes* { b2, b4, b6 }*. This means that these sets of blocks are* almost symmetrical.

Once *almost symmetrical* groups have been found, we can exploit these relationships in the following way. We can *prune* the search space as we have described above.

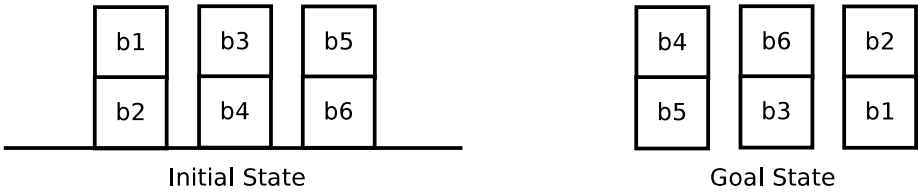

Figure 2.3: Example *Blocksworld* domain.

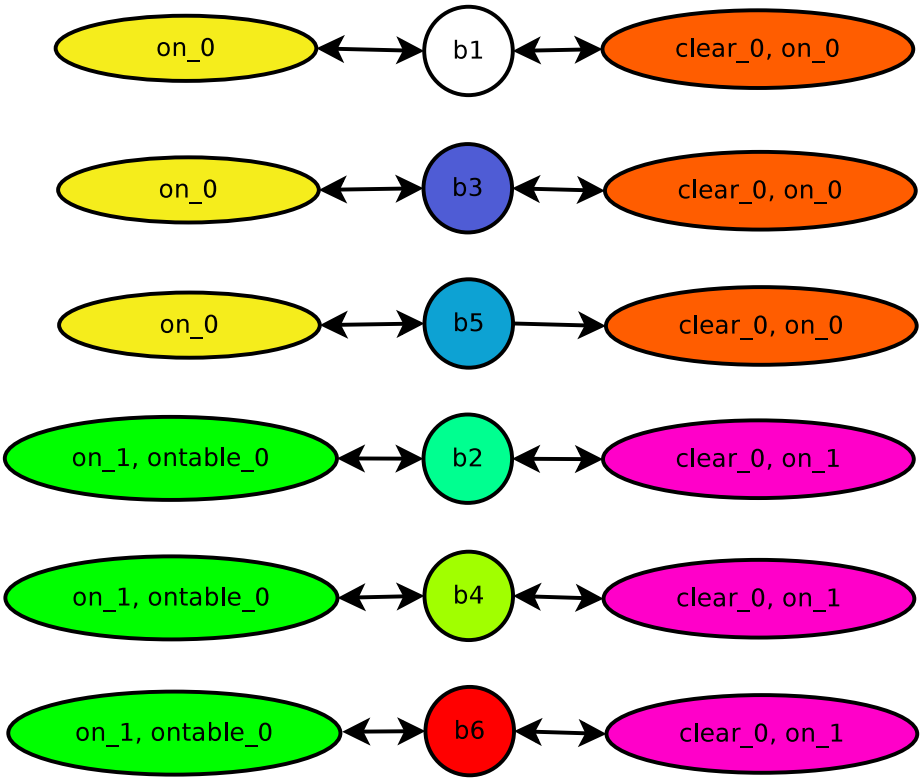

Figure 2.4: The graph created for the *Blocksworld* domain depicted in Figure 2.3.

This way of pruning is not solution preserving because it does not take the static constraints of a domain into account. For example, although two crates might be part of an *almost symmetrical* group because they are at the same location but on different pallets it might be the case that both pallets have different properties which necessitates vastly different sequence of actions to reach the goals.

We build upon the work carried out on almost symmetry but we address the shortcomings of the method used to find these *almost symmetrical* groups.

Chapter 3

Lifted relaxed planning graph heuristic

In this section we will show how we can adapt the h^{ff} heuristic without having to ground all the actions. We call this heuristic the *lifted relaxed planning graph heuristic* or h^{lrpg}. This is achieved by abstracting the objects in the planning problem. First we will explain how we use the analysis carried out by TIM to select which objects to abstract. Next we will describe the method used to construct a *lifted RPG* and show how it relates to the RPG. Next we describe improvements made to the implementation and data structures used to speed up the heuristic calculation. Finally we present empirical evidence of how our approach compares to the h^{ff} heuristic.

3.1 Motivation

Consider the construction of an RPG for a typed planning task Π using grounded actions. Recall that an RPG is constructed as follows:

Definition 37 — Relaxed Planning Graph

An RPG consists of alternating fact layers (fl) and action layers (al) which are constructed as follows. The first fact layer fl_0 consists of the set of literals which are true in s_0, then for $i \in \mathbb{N}_1$:

$$al_{i-1} = a \in A \mid a_{precs} \subseteq \mathit{fl}_{i-1}$$
$$\mathit{fl}_i = \mathit{fl}_{i-1} \bigcup_{a \in al_{i-1}} a_{effects^+}$$

We observe that the construction of the RPG entails a considerable amount of redundancy for most planning problems. This effect is aggravated as the planning problems grow in size.

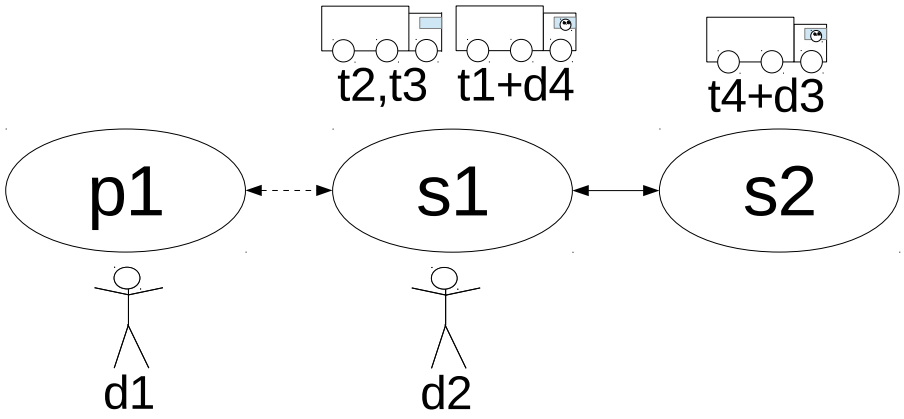

Figure 3.1: A simple Driverlog problem. The dashed lines can be traversed by drivers and the solid lines can be traversed by trucks.

Example 3.1.1 *Consider the Driverlog domain depicted in Figure 3.1. Each truck that starts at the same location and is empty can reach the same set of locations at each fact layer. Each location is reached by applying the same sequence of operators for each truck. For example, the truck* t2 *can reach the fact* (at t1 s2) *by applying the sequence* ⟨ (board d2 t2 s1)*,* (drive d2 t2 s1 s2) ⟩*. The truck* t3 *can reach the similar fact* (at t3 s2) *by applying the* equivalent *sequence of actions:* ⟨ (board d2 t3 s1)*,* (drive d2 t3 s1 s2) ⟩*. Furthermore, the same applies for any truck that does not start at* s1 *or is being driven, but* reaches s1 *and can become empty. These trucks are* equivalent*, so we can substitute these instances with a new object* T*. This reduces the size of the RPG considerably because instead of dealing with these two trucks separately we now only need to consider one.*

Just as the trucks in Example 3.1.1 can be made equivalent, it is clear that although the instances of type *driver* – *d1*, *d2*, *d3*, and *d4* – are not at the same location in the initial state they can become equivalent too. This is because all these drivers can *reach* the same locations and board the same trucks, any fact that is reachable for one driver is also reachable for the other drivers. In the next section we will formalise how we find the *equivalence* relationships between objects and how we subsequently exploit these relationships during the construction of the lifted RPG and the extraction of the heuristic.

In this work we find and exploit these *equivalence relationships* between objects and show the reduction in the number of actions we need to consider in order to construct the lifted version of the RPG. The concept of *equivalence* is tightly related to symmetry breaking, in particular *almost symmetry* [49].

3.2 Equivalent objects

As we have demonstrated in Example 3.1.1, we can significantly reduce the size and number of actions in an RPG if we combine objects into equivalence classes; We no longer need to include any actions that achieve facts which we know can be achieved due to symmetry breaking. Take another example. Consider a planning problem where we need to deliver a package and we have a fleet of identical trucks at our disposal. Instead of considering every truck (to deliver the package) we can *break* symmetry by realising that any of the trucks that reaches the target location of the package will suffice. We no longer need to generate the set of actions for any other truck which reaches the same location as the package, because the actions which are applicable to the truck that reached that location first are also applicable to any other truck which can reach that location.

We will now formalise the abstraction we apply.

Definition 38 — Object Equivalent Classes

Given a typed planning problem $\langle T, O, P, A, s_0, s_g \rangle$, a state s that contains all facts that are reachable from s_0 using the set of relaxed actions $\{a \in A \mid a_{effects} = a_{effects^+}\}$, two objects $o \in O$ and $o' \in O$ are part of the same equivalent class iff $Type(o) = Type(o')$ and $\kappa_{o,o'}(s) = s$, where

$$k_{o,o'} : state \rightarrow state,$$

such that $k_{o,o'}$ transposes all occurrences of o and o'.

If we apply Definition 38 to Example 3.1.1 then we can construct two *object equivalent classes*, one for all the drivers and one for all the trucks.

Definition 39 — Equivalent Objects

Given a typed planning problem $\langle T, O, P, A, s_0, s_g \rangle$, a state s, two objects $o \in O$ and $o' \in O$ are equivalent iff they are part of the same equivalent class and $\kappa_{o,o'}(s') \subseteq s_0$, where s' is the set of all facts in s that contain o or o'.

In Example 3.1.1 the trucks *t2* and *t3* are equivalent because they are at the same location and empty; Both can reach the same locations and can be driven by the same set of drivers by applying a similar sequence of actions. For example, consider any sequence of actions from the initial state – for example: { *(board d1 t2 s1)*, *(driver d1 t2 s1 s2)* } – if we alter the sequence of actions by replacing all instances of *t2* with *t3* then the sequence of actions is still valid and the resulting state is equivalent except that driver *d1* is driving the truck *t3* and *t3* is at location *s2*.

The drivers *d1* and *d2* – on the other hand – are not equivalent because they are both at different locations. However, if we execute the action *(walk d1 s1 p1)* or *(walk d2 p1 s1)* then both drivers become equivalent as well.

Object equivalence classes can be constructed by constructing an RPG till the level-off point and comparing the facts in the last fact layer using Definition 39. However, in order to construct the RPG we need to ground the domain.

In order to infer equivalence classes without having to ground the entire domain, we use TIM (see Section 2.6.1). TIM performs domain analysis on a typed planning task without having to ground the domain and infers a new type structure. Objects that

are part of the same set of *property spaces* and *attribute spaces* are assigned the same *type*. In Example 3.1.1, the objects *t1*, *t2*, *t3* and *t4* are assigned the same *type*. Using the analysis performed by TIM we construct the object equivalence classes as follows. Given an initial state s_0, an inferred type t, two objects $o \in t$ and $o' \in t$ are part of the same equivalence class iff the set of properties that are true for o in s_0 can be *exchanged* for the set of properties that are true for o' in s_0 using the transition rules of t and visa versa.

Example 3.2.1 *We refer back to the example depicted in Figure 3.1. Using TIM we can prove that the drivers* d2 *and* d4 *are part of the same object equivalence class. Both drivers are part of the inferred type that contains the objects* { d1, d2, d3, d4 }. *The set of properties that are true for* d2 *in the initial state is* $\{at_1\}$ *and the set of properties that are true for* d4 *in the initial state is* $\{driving_1\}$. *We can use the transition rule* $null \overset{board}{\Rightarrow} at_1 \rightarrow driving_1$ *to exchange the set of properties of* d2 *for the set of properties* $\{driving_1\}$. *Likewise we can reach the set of properties that are true for* d2 *in the initial state from the set of properties that are true for* d4 *in the initial state by using the transition rule* $null \overset{disembark}{\Rightarrow} driving_1 \rightarrow at_1$. *Therefore we can conclude that* d2 *and* d4 *are part of the same equivalence class.*

Following TIM's analysis we would conclude that all the drivers, trucks, packages, and locations are part of the same equivalent class for any possible Driverlog problem. However, this analysis only holds if the road network is connected, if the road network is disconnected then this analysis does not hold up. To demonstrate this we revisit example 3.1.1 but we make a small change to the road layout; imagine that there is no connection between *s1* and *s2*. In that case the two sets of trucks, { *t1*, *t2*, *t3* } and { *t4* } cannot become equivalent and are not part of the same equivalent class.

In order to differentiate objects that are part of the same type, as detected by TIM, but are not part of the same equivalent class we subdivide the detected types into subtypes. Using the TIM analysis we split up any type that contains a transition rule that gains or loses properties. In the above example we split up the inferred type that contains the objects { *s1*, *s2*, *p1* }, such that every object becomes part of a separate type. Now we can differentiate between objects that are part of different road networks. We further refine the types detected by TIM by comparing facts in the initial state that cannot be affected by any action, these are *static* facts. Let s be the set of static facts that are part of the initial state and contain the object o or o', o and o' cannot be part of the same type if $s \neq \kappa_{o,o'}(s)$.

Image that the Driverlog problem depicted in Figure 3.1 has the static fact *(is-small t1)* that allows the truck *t1* to access the location *p1*. In that case *t1* is no longer part of the same equivalence class as the other trucks, because the other trucks cannot access *p1*.

3.2.1 Reachability

We will now establish a relationship between equivalent objects and reachability, but before we do so we will first formally define what a *reachable atom* is.

Definition 40 — Reachable atom

Given a typed planning task Π and a state s, we say that an atom a is reachable if there exists a sequence of actions $\langle a_0, a_1, \ldots, a_n \rangle \mid \forall_{i \in \{0,\ldots,n\}} a_i \in \Pi_A$ such that $a \in \mathit{Result}(\langle a_0, a_1, \ldots, a_n \rangle, s)$.

Theorem 3.2.1 *Given a state s, a pair of objects o and o' which are equivalent in s and an atom $a = \langle p, V \rangle$ which is* reachable *from s, then $a' = \kappa_a(o, o')$ is also* reachable *from s.*

Proof: *Because a is reachable there must exists a sequence of actions $T = \langle a_0, a_1, \ldots, a_n \rangle \mid \forall_{i \in \{0,\ldots,n\}} a_i \in \Pi_A$ such that $a \in \mathit{Result}(\langle a_0, a_1, \ldots, a_n \rangle, s)$(see Definition 40). There are two cases to consider:*

- *In the simple case $\neg\exists_{v \in V} o \in D_v \vee o' \in D_v$. This means that $a \equiv \kappa_a(o, o')$.*
- *Otherwise there must exist a sequence of actions $T' = \langle a'_0, a'_1, \ldots, a'_n \rangle \mid \forall_{i \in \{0,\ldots,n\}} a'_i \in \Pi_A$, such that $a' \in \mathit{Result}(\langle a'_0, a'_1, \ldots, a'_n \rangle, s)$. This sequence is constructed as follows: every action $a'_i \in T'$ is identical to $a_i \in T$ except for its parameters.*

$$\forall_{\langle t, D_v \rangle \in a'_{i_{parameters}}} D_v = \begin{cases} (D_v \cup o) \setminus o' & \text{if } o' \in D_v \\ (D_v \cup o') \setminus o & \text{if } o \in D_v \\ D_v & \text{otherwise} \end{cases} \tag{3.2}$$

The preconditions and effects of each action in T' are updated accordingly. For every $a'_i \in T'$ the preconditions are defined as $a'_{i_{precs}} = \bigcup_{p \in a_{i_{precs}}} \kappa_p(o, o')$. Similarly the effects are defined as: $a'_{i_{effects}} = \bigcup_{e \in a_{i_{effects}}} \kappa_e(o, o')$.

We observe that if $s'_i = \mathit{Result}(\langle a'_0, \ldots, a'_i \rangle, s) \mid i \in \{0, \ldots, n\}$ is defined, then $s'_i = \bigcup_{f \in \mathit{Result}(\langle a_0, \ldots, a_i \rangle, s)} \kappa_f(o, o')$. If s'_i is not defined, then there must exist an i, such that $\operatorname{argmin}_{i \in \{0,n\}} \mathit{Result}(\langle a'_0, \ldots, a'_i \rangle, s) = \mathit{undefined}$. Then a'_i must have a precondition $p \in a'_{i_{precs}}$ that is not true in $s'_{i-1} = \mathit{Result}(\langle a'_0, \ldots, a'_{i-1} \rangle, s)$. However, we know that $\kappa_p(o, o') \in \bigcup_{f \in \mathit{Result}(\langle a_0, \ldots, a_i \rangle, s)} \kappa_f(o, o')$ because $\mathit{Result}(\langle a_0, \ldots, a_i \rangle, s)$ is defined. $\kappa_p(o, o')$ cannot be achieved by any of the actions $\langle a_0, \ldots, a_{i-1} \rangle$, because if there is an action $a_k \in \langle a_0, \ldots, a_{i-1} \rangle$ such that $\kappa_p(o, o') \in a_{k_{effects}}$ then $p \in a'_{k_{effects}}$. This means that $\kappa_p(o, o') \in s$ and $p \notin s$. This can only be the case if p and $\kappa_p(o, o')$ contain variable domains whose values contains o or o'; if this is true then o and o' are not equivalent.

Finally given the effect $e \in a_{n_{effects}}$ that makes a true, the last action a'_n has an effect equivalent to $\kappa_e(o, o')$ which makes a' true and completes the proof.

The above theorem does not imply that, given two equivalent objects o and o' and reachable facts f and $\kappa_f(o, o')$, both reachable facts can be true at the same time. However, if we were to ignore the delete effects of actions such as when we build an RPG f and $\kappa_f(o, o')$ can be true at the same time.

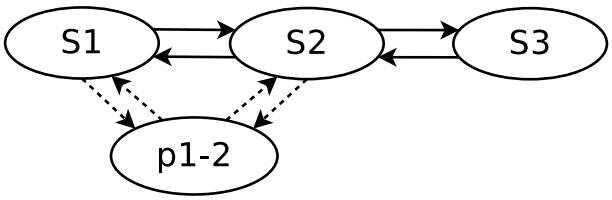

Figure 3.2: A small road network for the Driverlog domain.

To relate this definition back to discussion of *almost symmetry* (see Definition 36), we construct a coloured graph G where all the $o \in O$ objects in a typed planning task are added as a vertex, each with a unique colour. Next, for every object $o \in O$ we collect all the facts from the final fact layer $S \subseteq fl_n$, such that every fact $\langle p, V \rangle \in S$ contains a variable domain $\langle t, D_v \rangle \in V \mid o \in D_v$. Next we add S as a vertex to G; the colour is determined by the name, arity and the set of types of the predicate p and the set of indexes of o in each of the facts $s \in S$. Finally we add an edge between S and o All the possible symmetries of this graph based on the colourings yield the set of equivalent objects.

3.2.2 Which objects to ground?

As with the h^{ff} heuristic we construct an RPG, but we want to reduce the amount of grounding we need to perform. Given Theorem 3.2.1, if we only abstract objects that are part of the same *object equivalent class* then the lifted RPG will yield the same facts in the final fact layer as the RPG. By reasoning over sets of objects that are part of the same object equivalent classes instead of individual objects we can abstract the set of objects and significantly reduce the number of actions we need to ground.

Example 3.2.2 *Consider the road network depicted in Figure 3.2 for the Driverlog domain. Assume that there are five drivers, five trucks and five packages at each location.*

Using Definition 39 we can abstract all the drivers, trucks, and packages at the same location, which means that we only need: $4 * 4 * 4(board) + 4 * 4 * 4(load) + 4 * 4(walk) + 4 * 4 * 4(drive) = 208$ *actions. Compare this to the grounded case where there would be* $4{*}20{*}20(board){+}4{*}20{*}20(load){+}4{*}20walk{+}4{*}20{*}20drive = 4880$ *actions.*

However, we do not wish to find equivalence relationships between all possible objects of a planning problem. For example, objects that are part of atoms whose predicate is not affected by any operators in the domain preserve the *structure* of a planning problem, which will be reflected in the constructed lifted RPG. For example, consider the graphs depicted in Figure 3.3. The lines between the nodes depict a static predicate in a planning task (e.g. $(connected\ Loc1\ Loc2)$). In the case of a fully connected graph all location objects are equivalent. In the diamond-shaped graph $Loc1$ and $Loc2$ are equivalent, as are $Loc3$ and $Loc4$. Next we conclude that the equivalent pairs $(Loc1, Loc2)$ and $(Loc3, Loc4)$ are equivalent too so the entire graph is reduced to a single node and all location objects are equivalent.

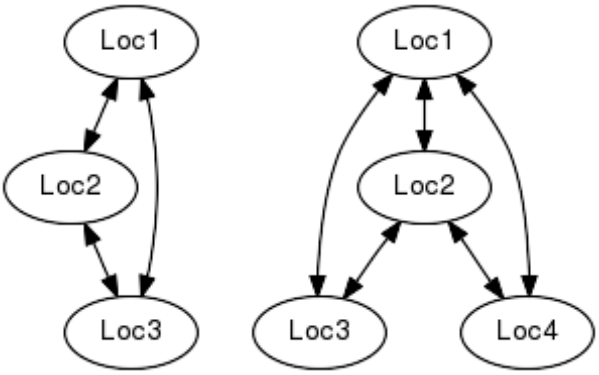

Figure 3.3: Two graphs where all locations are equivalent.

The example above highlights that we need to be careful about which objects we allow to be made equivalent. Objects of interest to us are part of *state invariants*, objects that are part of a set of predicates of which only one can be true at any given time (e.g. a truck can only be at a single location at any given time). Different methods have been developed to find these [2, 8, 27], and we use TIM [15] to infer the state invariants. We use the *Depots* domain from the IPC-3 [39] as a rolling example throughout this section and we assume the reader is familiar with this domain. In this domain crates are being transported between different places by trucks; at each location hoists can stack / unstack crates on top of pallets or other crates and load / unload crates from trucks.

3.3 The Lifted Relaxed Planning Graph

Consider the construction of an RPG for a typed planning task Π. If all the facts and actions are grounded then determining which actions are applicable to which fact layer is straightforward. However, if the facts are lifted then finding which operators are applicable becomes harder.

More specifically, given a set of lifted facts F and an operator o, we need to perform the following actions to find all the sets of all lifted facts which satisfy all the preconditions of o. We call each set of facts that satisfies all the preconditions of an operator *consistent*.

1. For every precondition $p \in o_{precs}$ we define the set of facts which *satisfies* p as $M_p = \{f \mid f \in F, f \textit{ satisfies } p\}$.
2. Given the sets of facts M_p – one for every precondition $p \in o_{precs}$ – we construct the Cartesian product over all satisfying facts. This yields $M_o = \{S \mid S \in \prod_{p \in o_{precs}} M_p\}$, where $\prod$ denotes the Cartesian product.
3. Finally we check which sets in M_o are consistent.

The construction and evaluation of the Cartesian sets is costly, especially if the set of facts F grows large. If an action is grounded then the sets of facts satisfying each precondition is either empty or contains a single element, which makes the process of finding consistent sets easy.

3.3.1 Partially Grounded Action

To reduce the computational overhead of constructing a lifted RPG we split the preconditions a_{precs} of an action a into disjoint sets P such that every element in $\prod_{p \in P} M_p$ is *consistent* with a. This reduces the overhead considerably. For example, assume we have split the preconditions into n sets each containing l preconditions and we can find m facts which can be assigned to each precondition; then we can find $m^l n$ possible consistent sets per action. By reducing l we can reduce the overhead of calculating the Cartesian products, which is the most expensive part of the algorithm. So it pays to split the set of preconditions into as many sets as possible. Compare this to the situation where the preconditions are not split, in which case there will be m^{ln} possible consistent sets per action. However, we need to be careful as the extreme case we go back to the grounded case where $m = l = 1$ and $n = \mid a_{precs} \mid$.

We must put constraints on how the preconditions are split in order to guarantee consistency. If a set of preconditions is split up into n disjunct sets then consistency can only be guaranteed if each precondition that shares a *variable* is in the same set or if each shared variable is grounded.

Example 3.3.1 *Consider the operator* (Load h - hoist c - create t - truck p - place) *with the following preconditions:*

- (at h p)
- (at t p)
- (lifting h c)

If these preconditions are split up as follows: $\{$(at h p), (lifting h c)$\}$ *and* $\{$(at t p)$\}$ *then we must* ground p. *If the predicate which is shared between the sets –* p *– is not grounded, we could find the following consistent sets:* $\{$(at hoist1 depot0), (lifting hoist1 crate4)$\}$ *and* $\{$(at truck8 distributor4)$\}$ *whose product is not consistent because the hoist and trucks are at different locations.*

The preconditions in the example above can be split in many ways. We want to limit grounding as much as possible, but want to limit the time necessary to find all consistent sets as well. As discussed in the introduction we want to reduce the redundancy in constructing the lifted RPG by abstracting equivalent objects. Therefore we limit grounding to those objects which cannot become equivalent. Using the analysis performed by TIM we use the following rules to determine how to split a set of preconditions.

Proposition 3.3.1 *Given a typed planning problem* Π *and the set of objects* O' *which are part of at least one* property state *then the objects are divided into disjunct sets as*

follows. Given any two objects $o, o' \in \Pi_O$ then o and o' will be part of the same set iff *the following properties are true:*

- *o and o' are of the same* type*.*
- *$o \cap O' \neq \emptyset$ and $o' \cap O' \neq \emptyset$.*
- *For every* static *fact $f \in \Pi_{s_0}$ there exists an equivalent fact $\kappa_f(o, o')$ that is also true in Π_{s_0}.*

If these conditions hold for any two objects o and o' then they can potentially *become equivalent and thus are stored in the same set.*

Proposition 3.3.1 creates a set of sets of objects per type. We ground all the actions such that the domain of every parameter is equal to one of the sets of objects according to the type of the parameter.

Example 3.3.2 *Consider the* drop *operator from Example 3.3.1. Assume there are three places* depot0*,* depot1*, and* depot2*. Each place has two hoists, there are ten crates distributed among the places and there are three trucks. The following partially grounded actions are constructed for every place $p \in$ {depot0, depot1, depot2}: (Load {p-hoist0, p-hoist1} {crate0, crate1, ..., crate9} {truck0, truck1, truck2} p). In this instance we end up with three partially grounded actions – one for each place – instead of 180 grounded instances (assuming that (partially) grounded actions whose preconditions do not satisfy the static facts in s_0 are not constructed).*

Given the set of partially grounded actions we use the following procedure to split the preconditions up into separate sets. Initially we pick any precondition at random and add it to a new set P. Next we add any precondition not part of a set yet but which shares a parameter with another precondition in P whose domain contains more than one object. If no more preconditions can be added to P, start a new set and follow the same steps. Continue until all preconditions are added to a set.

Example 3.3.3 *In our example we end up with the following sets of preconditions for every place $p \in$ {depot0, depot1, depot2}:*

- *(at { p-hoist0, p-hoist1 } p), (lifting { p-hoist0, p-hoist1 } { crate0, crate1, ..., crate9 }).*
- *(at { truck0, truck1, truck2 } p).*

Definition 41 — Partially Grounded Action

Given a planning problem Π and action $a \in \Pi_A$ then a *partially grounded action* is a tuple $\langle name, parameters, precs, effects \rangle$ that is identical to a except for $precs$ and $effects$, which are sets of disjunct sets of facts as defined above.

All problem domains on which we have tested this approach only ever required one or two sets to split all the preconditions.

3.3.2 Merging

After splitting the preconditions into separate sets we notice that many of the sets are identical. By merging sets which are equivalent we can reduce the overhead of finding consistent mappings of facts for the same set of preconditions.

Definition 42 — Set equivalence

Two sets of preconditions are equivalent if there exists a bijection of both sets of atoms where the predicate and variable domains of each mapped pair are the same. In addition, two variables of any pair of atoms are equal in one set if and only if the variables corresponding to the mapped atoms are also equal.

The latter requirement of Definition 42 is necessary for some domains where – based on the predicate and variable domains alone – multiple bijections are possible. One such example is the Blocksworld domain: some sets contain the pair $(\text{on block block}) \wedge (\text{on block block})$ so the relationships between the variables become important to create the correct bijection. For any pair of equivalent sets we update the partial grounded action in such a way that for any set of equivalent sets only one is used to find consistent sets of facts.

To decrease the overhead even further we found that for some actions the set of add effects is equivalent to some precondition sets. For each action we create a new set of facts that contains all the add effects and all preconditions which were not removed, and split them using the same process we used to split up the preconditions.

Example 3.3.4 *The* Drop *operator from Example 3.3.1 yields the following sets for every place $p \in \{$depot0, depot1, depot2$\}$:*

- *(at { p-hoist0, p-hoist1 } p), (available { p-hoist0, p-hoist1 }).*
- *(at { truck0, truck1, truck2 } p), (in { truck0, truck1, truck2 } { crate0, crate1, . . . , crate9 }).*

These are exactly the preconditions for the operators Lift *and* Unload.

This completes the construction of the partially grounded actions. Comparing the number of partially grounded actions to all the grounded actions we notice that in the worst case we define exactly as many actions, but in most cases the number of partially grounded transitions is a multitude of magnitudes smaller. Table 3.1 records the number of transitions for the largest problems for some domains used in various international planning competitions.

3.3.3 Creating the lifted relaxed planning graph

We will now explain how we construct a lifted RPG. The pseudo code is shown in Algorithm 3, where $EO : objects, \mathbb{N} \rightarrow objects$ is a function that maps a set of objects to the set of objects that are *equivalent* relative to the set of facts in a fact layer. For example, $EO(O, n)$ maps the set of object O to the equivalent set of objects that are equivalent to O in the nth fact layer. We shall now discuss every step in greater detail.

	Zeno	Satellite	Storage	Blocksworld
Lifted transitions	5500	4562	170	4
Grounded Transitions	959530	43290	348660	612
	Driverlog	Gripper	Depots	Rovers
Lifted transitions	528	8	112	7364
Grounded Transition	218300	6204	55936	423064

Table 3.1: Reduction of the number of transitions.

The starting point of constructing the lifted RPG is the set of atoms in the initial state. Next we establish all equivalent relationships between objects and copy them to the subsequent fact layer using NOOP actions. Finally, given the current fact layer fl_i and a partially grounded action a, for each set of preconditions $P \in a_{precs}$ we search for all consistent sets $M_P \subseteq \prod_{p \in P} M_p$. Next we take the Cartesian product over all these consistent sets $C = \prod_{P \in a_{precs}} M_P$ to get all the consistent sets for all the preconditions of a. We apply the action a to every consistent set $c \in C$ and add every effect $e \in a_{effects}$ to the next fact layer and record a as its achiever. We denote all the achievers of every lifted fact l as $achievers(l)$.

Even though we have split all the preconditions into disjunct sets, the computation of the Cartesian sets is still expensive. We use a tree structure to compute the Cartesian sets iteratively. Given a partitioned transition a and a set of preconditions $P \in a_{precs}$, then the roots of the trees are all the facts which satisfy the first precondition $p_0 \in P$. The nodes at depth i are the facts which can satisfy the precondition $p_i \in P$. A tree is constructed as follows. First we construct the root after a fact is added which satisfies p_0. Next we search for facts which satisfy $p_i \mid i \in \{1, 2, \ldots, \mid P \mid\}$ and add them as children to p_{i-1}, but only if the conjunction of all the nodes along the *path* from the node added at p_i to the root of the tree is *consistent* with P. The benefit is that we have to evaluate fewer potential consistent sets and use less space to represent them. For example, if a new fact f is added that is consistent with the last precondition $p_{|P|-1} \in P$ and every variable domain of $p_{|P|-1}$ is shared with p_0, then we only need to evaluate the root nodes of every tree to check in which tree f can be added as a leaf node.

Example 3.3.5 *As an example of what a lifted RPG looks like after construction, consider the Driverlog domain depicted in Figure 3.4. We have two drivers who can walk over the path* s2 $\leftrightarrow$ p1-2 $\leftrightarrow$ s1 $\leftrightarrow$ p1-3 $\leftrightarrow$ s3 *and board the only truck in the problem located at location $s3$. The goal of the domain is to achieve the fact* (driving d2 t1)*; the constructed lifted RPG is depicted in Figure 3.5. Note that we have removed all the static facts, such as the connection between the paths, and only list the first achiever of any fact for readability.*

Till fact layer 2 the lifted RPG is identical to that of the grounded RPG. However, when reaching fact layer 2 we can make both drivers equivalent because both drivers have reached the initial location of the other. Thus in fact layer 2 the drivers can be used interchangeably, which means that we now know that driver $d2$ can reach the fact $(at\ d2\ s3)$ even though no action in action layer 1 achieves this fact. In general this

Algorithm 3: Constructing a Lifted Relaxed Planning Graph till Level-Off Point.

$fl_0 = s_0$;
for $i \in \mathbb{N}_1$ **do**
  // Update the facts relative to the equivalent objects and delete any duplicates.;
  $fl_i = \{f \mid f \in fl_{i-1}\}$;
  for $f = \langle p, V \rangle \in fl_{i-1}$ **do**
    $f' = \langle p, V' \rangle \mid V'_i = \langle V_{i_t}, EO(V_{i_D}, i-1) \rangle$;
    if $f' \notin fl_i$ **then**
      $fl_i = fl_i \cup f'$;
  for $a \in PartiallyGroundedActions$ **do**
    // Find all consistent sets of facts in the current fact layer;
    for $P \in a_{precs}$ **do**
      for $p \in P$ **do**
        $M_p = \{f \mid f \in fl_{i-1}, f \text{ satisfies } p\}$;
      $M_P = \{s \mid s \in \prod_{p \in P} M_p\} \mid$ s is consistent with P;
    // Take the product of all consistent sets of facts;
    for $C \in \prod_{P \in a_{precs}} M_P$ **do**
      //Instantiate the action with the consistent set of preconditions C;
      $fl_i = fl_i \cup \bigcup_{e \in a_{effects}} e$;
      $al_{i-1} = al_{i-1} \cup a$;
  if $fl_i \equiv fl_{i-1}$ **then**
    break;

means that the number of fact layers in the lifted RPG will at most be equal to that of a grounded RPG but more often than not contains considerable fewer.

3.4 Performing reachability analysis

Apart from using the RPG for computing heuristics, it is also used to check for dead ends. Any facts that do not appear in the final fact layer of an RPG that is generated till the level off point are not reachable. If any of these facts are a goal than we have reached a dead end. In the previous section we have explained how a lifted RPG is constructed; in this section we will examine how well our approach scales compared to the grounded approach. For our analysis we use Marvin [6] which participated in the fourth international planning competition [31].

One important aspect of a typed planning problem is how many objects can be proved to be equivalent, as this allows us to prove facts to be reachable for whole sets of objects. If very few equivalent relationships can be proved we pay a lot of overhead without benefiting from it. In the worst case scenario no equivalent relationships can be found; in this case we graciously fail and the constructed lifted RPG is identical to the grounded RPG. We define the *compressed size* as the ratio of the number of sets

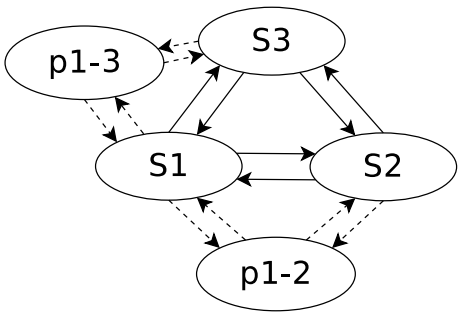

Figure 3.4: Driverlog example domain.

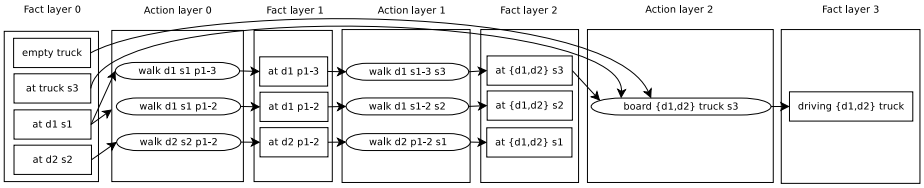

Figure 3.5: Driverlog lifted RPG.

of equivalent objects considered by the lifted reachability analysis to the total number of objects in a problem domain. We should expect the compressed size to be inversely proportional to the relative performance of our lifted algorithm. The smaller the compressed size, the more benefit we expect over the grounded approach. This trend is clearly visible in Figure 3.6. Table 3.2 shows the compressed size for the biggest problem instances of the IPC competition domains and the number of sets of equivalent objects considered by our algorithm during every iteration.

The time taken results are shown in Figure 3.7. The domains Gripper, Satellite, Zeno, Depots[1] and Blocksworld support our hypothesis. Gripper is an example which

[1]We debugged the domain definition by enforcing the condition that the surface a crate is lifted from is at the same location.

Domain	Objects	Sets of equivalent objects per iteration	Compressed size
Rovers	61	58/58/58/58/58/58	95%
Storage	52	52/42/41/41/41/41/41	79%
Driverlog	99	83/78/63/63	63%
Depots	46	44/39/32/27/27	58%
Zeno	60	49/43/32/32	53%
Blocksworld	18	4/2/2	11%
Satellite	70	8/6/6	9%
Gripper	781	6/6/6/6	1%

Table 3.2: The compressed size of the largest problem instance per domain.

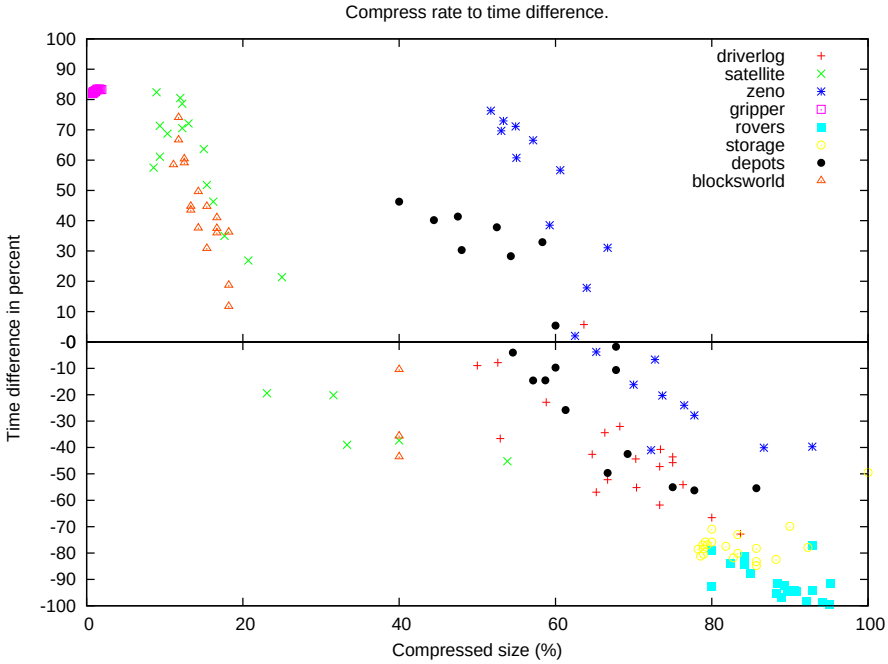

Figure 3.6: The relative performance compared to the compressed size.

has been proved susceptible to other symmetry related techniques [47, 49] and ours is no exception. The effects on Zeno, Blocksworld and Depots, on the other hand, are not as obvious. The small problem instances do not show any benefit, but when the problems become bigger and more objects can be made equivalent we see the benefit of lifting.

Storage and Rovers are clearly not very suitable for our lifted approach. In the case of Rover this is not surprising because the problem instances describe Rover objects which all have different properties, e.g. different sets of waypoints they can traverse and different configurations of instruments at their disposal, thus very few objects can be made equivalent. In these instances we pay for the overhead during propagation and trying to establish new equivalence relationships but do not benefit from it. This is clearly depicted in Figure 3.6. Storage offers a little more scope, but the domain does not yield many objects which can be made equivalent due to the different static relationships all these objects have to each other.

Depots and Driverlog are the most interesting domains because their performance is on a par with the grounded approach. The lifted approach seems to become preferable for the Driverlog domain as the domain becomes bigger, while for the Depots domain the compressed size seems to be more important than problem size. To test if our approach scales better, we have generated twenty larger problems for a subset of the domains, using the generators used at the competition by multiplying the number of objects from the original problems by ten. For Blocksworld we used a publicly

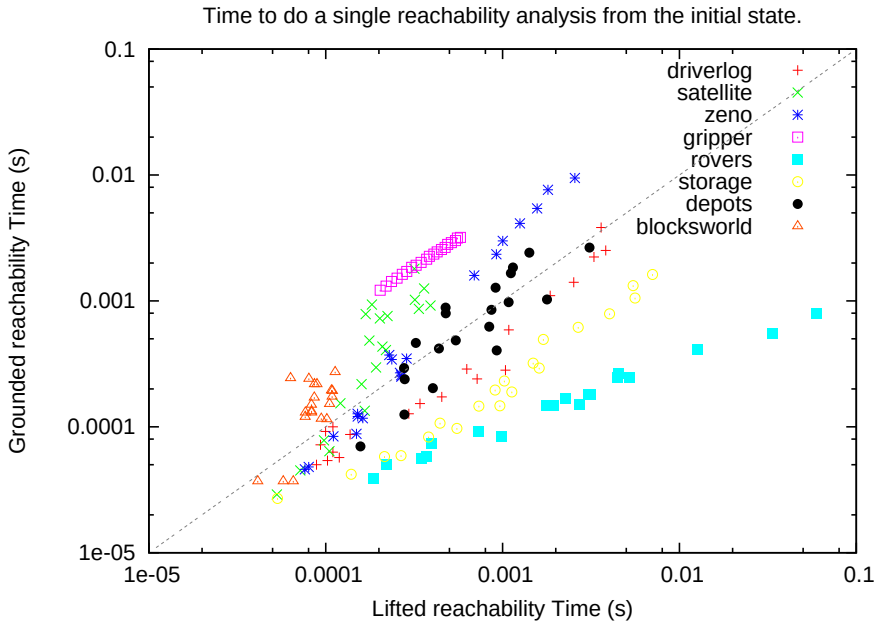

Figure 3.7: Comparing the grounded versus the lifted approach.

available generator[2] and we generated problems containing a multiple of 100 blocks. The results are shown in Figure 3.8.

Our approach clearly scales better than the grounded approach, which could not perform a reachability analysis for all problem files of any domain, except for Storage, because it ran out of memory. The grounded approach is still faster on the two smallest Depots problems but fails on the larger problems. This again can be contributed to the large compressed size of these problems, 71% and 62% respectively. The same is true for the Storage problems where the compressed size fall between 76% and 78%. We have experimented with Storage problems with higher compressed size (up to 56%) and found that, although the relative performance improves, the grounded approach is still more favourable.

3.5 Calculating the heuristic

Having established how a lifted RPG is constructed we will now explain how to calculate the heuristic h^{lrpg}. To determine how informative this heuristic is we compare it to the FF heuristic h^{ff} [30]. Like the FF heuristic we extract a relaxed plan from the lifted RPG. However, due to the object abstraction there are some key differences in how we calculate our heuristic compared to h^{ff} which we will describe in this section.

Given a planning problem $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ we begin by constructing a

[2] `http://www.loria.fr/~hoffmanj/ff-domains.html`

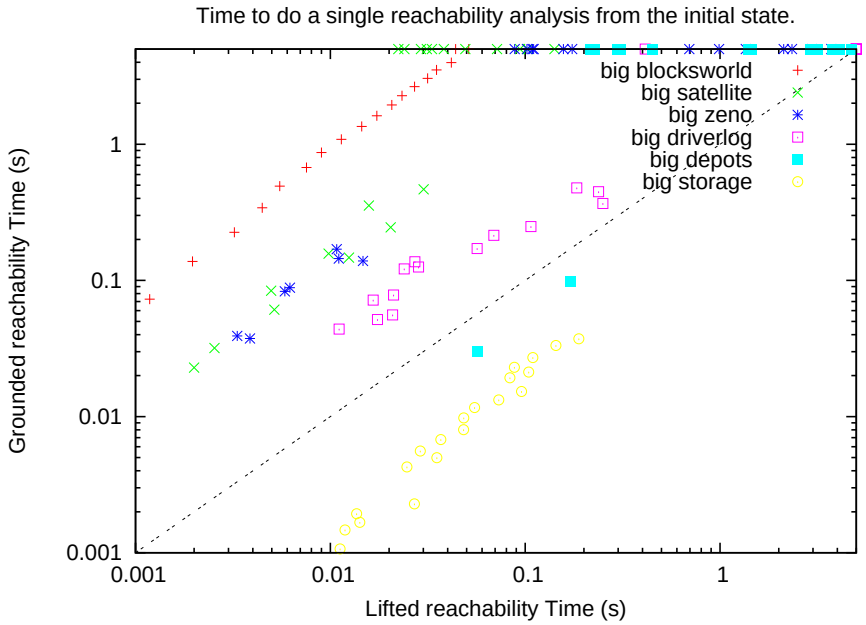

Figure 3.8: Results for the larger domains.

lifted RPG until the level-off point. We do not construct the lifted RPG until we construct a fact layer fl_i such that for every goal $s_g \subseteq fl_i$. The reason why we construct the complete lifted RPG will become apparent once we discuss how we select achievers, find *helpful actions* and how we make *substitutions*. Once the lifted RPG is constructed we add all the lifted facts f which correspond to the goal s_g to an open list G. G contains all the facts which have to be achieved; once G is empty we have found a relaxed plan which solves the relaxed planning problem.

3.5.1 Naive Fully Lifted RPG heuristic algorithm

As discussed in Section 3.3.3 the lifted RPG will at most contain as many fact and action layers as the grounded RPG, but for most planning problems it will create considerably fewer. While this allows us to create the lifted RPG faster than the RPG and exempts us from having to ground the domain, it also affects the heuristic estimate we can find. We will first introduce an algorithm which mimics h^{ff} as closely as possible and discuss the problems with this algorithm and introduce improvements to this naive algorithm. We think this is important in order to understand the strengths and limitations of this algorithm.

The naive lifted RPG heuristic is calculated as follows. While G is not empty, we remove a fact l that is part of a fact layer fl_i from the open list G such that there is no other fact $l' \in G$ that is part of fact layer fl_j, such that $j > i$ and select an achiever $a \in achievers(l)$. If more than a single achiever is present we select the achiever

whose preconditions are estimated to be easiest to achieve. This cost is estimated by $action\ cost(a)$.

Definition 43 — Action Cost
Given an achiever a in action layer al_i the *action cost* is defined as:

$$action\ cost(a) = \Sigma_{p \in a_{precs}}\, cost(p, i), \tag{3.3}$$

where $cost(p, i)$ is defined as

$$cost(p, i) = \begin{cases} 0 & \text{if } i = 0 \\ i & \text{if } NOOP \notin achievers(p) \\ cost(p, i-1) & \text{otherwise} \end{cases} \tag{3.4}$$

We select the achiever a with the lowest cost. If a has not already been added to the relaxed plan, then we add a to the relaxed plan and add all the preconditions of a to G. We continue this process until G is empty. The heuristic estimate is the length of the relaxed plan.

This heuristic is very fast to calculate. This is because the time to construct the lifted RPG is lower for most domains compared to the grounded RGP (see Section 3.4) and because the lifted RPG contains *at most* as many action layers, fact layers, actions, and facts compared to the grounded RPG. The method we use to extract a relaxed plan is equivalent to that of h^{ff}, so the heuristic estimate will always be equivalent to or lower than the heuristic estimate of FF.

While we use far less memory than FF to calculate a heuristic, which makes this heuristic estimate suitable for larger problem instances, its heuristic estimate is not as informative as h^{ff}. This is to be expected since we treat objects as equivalent and introduce shortcuts in the relaxed plan.

Example 3.5.1 *Consider Example 3.3.5 in the case where we want to achieve the facts:* (driving d2 t) *and* (at d1 s3). *First we look for an achiever for* (driving d2 t). *In the lifted RPG there is only a single action which achieves* (driving d2 t)*, which is* board {d1, d2} t s3. *We add this action to the relaxed plan. Next we add all the preconditions to G. These are:* (at t s3) *– which is part of the initial state – and* (at {d1, d2} s3). *However, the latter precondition is already a goal in the plan because* (at d1 s3) *was added first.*

Next we add the actions to achieve goal (at {d1, d2} s3) *to the relaxed plan. In this case there is only a single achiever which is* (walk d1 s1 p1-3).*We add the precondition* (at d1 p1-3) *to G which is solved by adding the action* (walk d1 s1 p1-3) *to the relaxed plan. So the final relaxed plan is the following:*

- (walk d1 s1 p1-3).
- (walk d1 p1-3 s3).
- (board {d1, d2} t s3).

Compare this to the relaxed plan constructed by FF, which is:

- *(walk d1 s1 p1-3).*
- *(walk d1 p1-3 s3).*
- *(walk d2 s2 p1-2).*
- *(walk d2 p1-2 s1).*
- *(walk d2 s1 p1-3).*
- *(walk d2 p1-3 s3).*
- *(board d2 t s3).*

The lifted RPG makes $d1$ and $d2$ equivalent, which means that they become indistinguishable from each other. We would construct the same lifted plan if *(driving d2 t)* was the only goal to achieve. In that case *helpful actions* are not that helpful. For example, if the goal to achieve is *(driving d2 t)*, then the only helpful action would be *(walk d1 s1 p1-3)* which does not get us closer to the goal. Instead, we need to realise that the *identity* of the object that boards truck t and needs to get to location $s3$ is driver $d2$ instead of $d1$. In order to compensate for the change of identity we need to annotate the heuristic function with the cost of *substituting* $d2$ with $d1$. To do this we change the way we construct the relaxed plan in order to detect whenever we need to make a substitution.

3.5.2 Enhanced Fully Lifted RPG heuristic algorithm

As can be seen from Example 3.5.1 while the goal is to get $d2$ into the truck the relaxed plan only contains actions which move driver $d1$ towards the truck. In order to account for these discrepancies we select achievers differently from the naive method. Instead of picking the achiever with the lowest action cost to achieve a fact $l = \langle p, V \rangle$, we first check for each possible achiever a if it necessitates *substitution*.

Definition 44 — Substitution

Given an action a that contains an effect $e = \langle p, V' \rangle$ that has been selected to achieves a goal $f = \langle p, V \rangle$ then we need to make a substitution *iff* there exists an index $j \in \{0, 1, \ldots, \mid V \mid -1\}$ such that, given the variable domains of the effect $V_j = \langle t, D \rangle$ and the variable domains of the goal $V'_j = \langle t, D' \rangle$, $D \cup D' = \emptyset$. In that case we say we need to make a substitution between the object sets D and D'.

A substitution can also arise if the variable domains of a precondition $r = \langle p, V \rangle \in a_{precs}$ do not match up with the variable domains of a fact $f = \langle p', V' \rangle$ from a fact layer it is linked to.

The naive lifted RPG heuristic algorithm does not detect when substitutions need to be made. In this section we describe two methods that identify which substitutions need to be made. The first algorithm will limit the number of actions that can be added to the relaxed plan to the number of actions in the lifted RPG and introduces a small modification to the naive algorithm. The second algorithm uses the actions in the lifted RPG as a *template* from which to create more actions to the relaxed plan. The latter

method will detect more substitutions than need to be made, but also has a heavier memory requirement – albeit not as big as any grounded approach. In addition we will discuss two methods to handle *substitutions* and introduce some novel pruning techniques to speed up the search process.

First of all we will describe a small modification made to the naive algorithm that allows us to detect when substitutions need to be made. We will discuss how substitutions are handled in the next section.

Given the selected achiever a and effect $\langle p, V \rangle \in a_{effects}$ to achieve $l = \langle p, V' \rangle$, we update the variable domains of a so that V_i is equal to the intersection of V_i and V_i', unless this intersection is empty. If this intersection is empty, then we need to make a *substitution* between the objects which are part of V_i and those that are part of V_i' and we do not update the variable domain V_i. Next we check the updated variable domains of the action with the preconditions in the previous fact layer. If the intersection with the variable domains of the preconditions and the updated variable domains of the action is empty then we need to make another *substitution*.

If more than a single achiever is available to achieve a fact l then we prefer the achievers that do not need to make any substitutions. This is why we construct the lifted RPG till the level-off point. Whilst a goal $l \in G$ can be part of fact layer fl_i, it might be the case that there exists a sequence of actions found at fact layer fl_j, where $j > i$ that does not necessitate any substitutions and gives a better heuristic estimate than the sequence of actions found from the matching fact in fl_i. However, there is no guarantee that any sequence of actions exists which does not necessitate substitutions. For example, the lifted RPG constructed from Example 3.5.1 does not contain any sequence of actions that will not require any substitutions because $d1$ and $d2$ become equivalent before $d2$ reaches the location of the truck $s3$.

Example 3.5.2 *Returning to Example 3.5.1, if we use the above technique to detect substitutions we will not find any other relaxed plan. The reason is that when we select* (board {d1, d2} t s3) *as an achiever for the goal* (driving d2 t)*, we update the variable domains of the achiever accordingly which yields* (board d2 t s3) *that does not necessitate any substitutions. When we compare the preconditions of this achiever with the facts in the previous fact layer we find that the precondition* (at t s3) *maps to the fact* (at t s3)*, so no substitutions are required there and the precondition* (at d2 s3) *maps to the fact* (at {d1, d2} s3) *which also does not require any substitutions. So the relaxed plan will be identical to that found in Example 3.5.1.*

However, if the initial state is such that $d1$ is at location $s3$ then we do detect the need to make a substitution. The lifted RPG is depicted in Figure 3.9. The naive algorithm would find the relaxed plan:

- (board d1 truck s3).

If we use the enhanced algorithm, then we would find the same relaxed plan. However, when we check if we need to make substitutions then we find that when we select the NOOP to achieve (driving d2 truck s3) *that the precondition of that NOOP does not match the variable domain of the effect. The precondition is* (driving d1 truck s3)*, so here we find that we need to make a substitution between $d1$ and $d2$.*

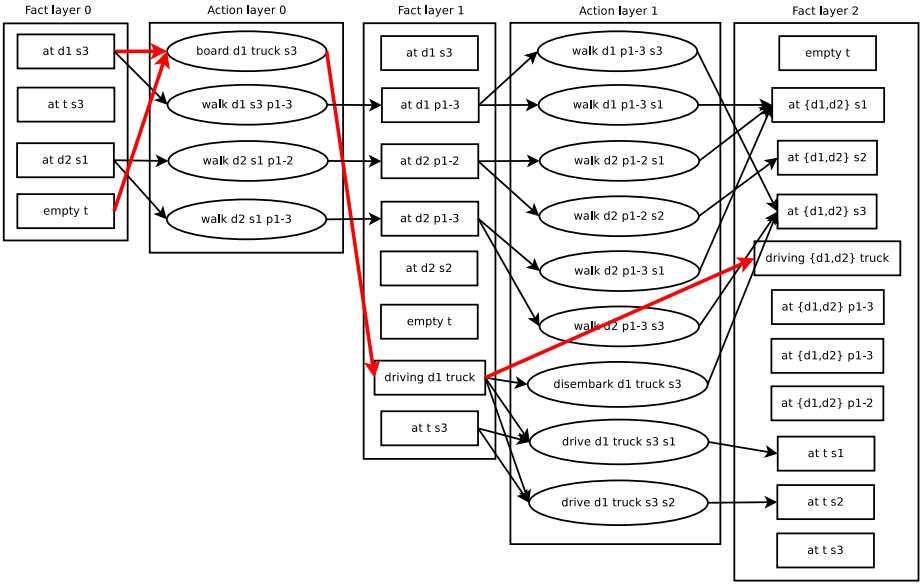

Figure 3.9: Driverlog lifted RPG after moving $d2$ to $s3$. NOOPs have been excluded from the graph for readability, except the NOOP which is selected as an achiever. The actions selected to add to the relaxed plan are marked red.

The example above highlights that while there are situations where we can improve the heuristic estimate by finding substitutions it does not always work, as shown by applying this algorithm to Example 3.5.1.

3.5.3 Enhanced Partially Lifted RPG heuristic algorithm

For the final algorithm we relax the constraint that we can only add as many actions to the relaxed plan as there are actions in the lifted RPG. Whenever we select an achiever a to achieve a goal $l = \langle p, V \rangle \in G$ then we first check if there exist an action in the relaxed plan that is an copy of a such that there exists an effect $e = \langle p, V' \rangle \in a_{effects}$ such that e achieves l. If such an achiever exists then we update the variable domain as follows.

Given the variable domains $V_i = \langle t, D \rangle$ and $V_i' = \langle t', D' \rangle$, where $i \in \{0, 1, \ldots, \mid V \mid -1\}$ then we update each domain $D_j \in V_i \mid j \in \{0, 1, \ldots, \mid D \mid\}$ such that $D_j = D_j \cap D_j'$.

If no copy of a can be found than we create a copy and update the variable as described above. This method finds better heuristic estimates, both because an achiever can be added multiple times to the relaxed plan so we are not bound to the number in the relaxed plan, and because at the same time it detects more substitutions that need to be made.

Example 3.5.3 *Consider Example 3.3.5 in the case where we want to achieve the facts:*

(driving d2 t) *and* (at d1 s3). *First we look for an achiever for* (driving d2 t). *In the lifted RPG there is only a single action which achieves* (driving d2 t)*, which is* board {d1, d2} t s3. *There are no actions in the relaxed plan so we create a new action to put into the relaxed plan. We update the variable domains of the selected action and check if we need to make any substitutions. In this case we do not have to make any substitutions and the action to be added to the relaxed plan is:* (board d2 t s3). *Next we add all the preconditions to G. These are:* (at t s3) – *which is part of the initial state – and* (at d2 s3).

The next goal is (at d2 s3). *Note that the fact in the second fact layer is represented as* (at driver1, driver2 s3) *and can only be achieved by a single achiever which is* (walk d1 p1-3 s3). *Because we updated the action domain variables we end up with a discrepancy between the effect of the only achiever which is* (at d1 s3) *and the original goal. So we need to resolve the discrepancy between the two objects* d2 *and* d1. *We will discuss how to handle substitutions below and ignore it for this example. We add the precondition* (at d1 p1-3) *to G which is solved by adding the action* (walk d1 s1 p1-3) *to the relaxed plan.*

Lastly we solve the last remaining goal (at d1 s3)*; as when trying to solve the goal* (at d1 s3) *there is only a single achiever which is* (walk d1 p1-3 s3). *Instead of adding a new action to the relaxed plan we check if we can update the variable domain of the already executed action to achieve this goal. In this case we can, so we do not need to add any other actions to the relaxed plan. So the final relaxed plan is the following:*

- (walk d1 s1 p1-3).
- (walk d1 p1-3 s3).
- (board d2 t s3).

The above example shows that the extraction of a plan is reminiscent of the process FF uses to extract a relaxed plan. The main differences are that (1) an action in an action layer can be *instantiated* multiple times, each time with different values of their variable domains; and (2) the relaxed plan may not be valid, for example the relaxed plan in Example 3.5.3 is not valid because we need to *substitute d2* with *d1*.

There is an inherent problem using this method, because in the worst case scenario we need to ground all actions. This means that this method might will not do any better than other grounded approaches. However, while these planning problems exist we have found that for the benchmark domains on which we have tried this method we still use significantly less memory than any grounded approaches.

3.5.4 Substitutions

Due to the way we construct the lifted RPGs it is possible that extracted relaxed plans are not valid. The need for a substitution arises whenever we add an action to the relaxed plan to satisfy a goal $l \in G$, but (1) the variable domains of the effect of the selected achiever of l do not match up with the variable domains of l or (2) when we select an achiever for l and update the variable domains of the achiever accordingly, such that we find a discrepancy between the variable domains of a precondition of the

achiever and the actual facts in the fact layers. The latter occurs in Example 3.5.3. We need to account for the discrepancies between the variable domains of the effects and / or preconditions and the corresponding facts in the fact layers to get a better heuristic estimate. Effectively we allow the relaxed plan to take *shortcuts* and need to account for these.

In order to account for any substitutions that need to be made, we present the following two methods:

- **ObjectSub**: Given a fact $f = \langle p, V \rangle$ that needs to be achieved and a precondition $f' = \langle p, V' \rangle$ from the achieving action, for every pair of variable domains $D \in V$ and $D' \in V'$ that are related to the same action parameter and whose intersection is empty we find the first fact layer where $o \in D$ and $o' \in D'$ becomes equivalent and add the layer number to the total of the heuristic. Unless we have already made a substitution between o and o', in that case we do not alter the heuristic.

- **GoalSub**: Given a fact $f = \langle p, V \rangle$ that needs to be achieved and a precondition $f' = \langle p', V' \rangle$ from the achieving action, we construct a new fact $f'' = \langle p', V'' \rangle$, where every variable domain $\langle t'', D_i'' \rangle \in V_i''$ is equal to $D_i' \in V_i'$ iff the action parameter V_i' is related to is not related to any variable domain $v \in V$, otherwise it is equal to the intersection between D_i' and the variable domain $D \in V$ that is related to the same action parameter. If that intersection is empty then D_i'' is equal to D. This new fact f'' is added to G.

The pseudocode of extracting a relaxed plan using substitutions is shown in Algorithm 4.

Algorithm 4: Extract a Relaxed Plan

Data: The set of fact layers fl, the set of action layers al, the initial state s_0, the goal state s_g, and the subsitution method M (if any).
Result: The relaxed plan T.
begin
  $G \longleftarrow \{\langle f, \mid \mathit{fl} \mid \rangle \mid f \in s_g\}$;
  while $G \neq \emptyset$ **do**
    $\langle f, i \rangle \in G \mid \neg\exists \langle f', j \rangle \in G \mid j > i$;
    $G \longleftarrow G \setminus \langle f, i \rangle$;
    $a \in al_{i-1} \mid a$ is the cheapest achiever for f;
    if $M \in \{\mathit{ObjectSub}, \mathit{GoalSub}\}$ **then**
      Augment the heuristic estimate and add facts to G according to M;
    if $a \notin T$ **then**
      $T \longleftarrow T \cup a$;
    if $i - 1 \neq 0$ **then**
      $G \longleftarrow G \cup \langle p, i - 1 \rangle \mid p \in a_{precs}$;
end

Whenever a substitution is made between two sets of objects it is recorded so that we do not make the same substitution multiple times.

We do not use the latter method if the achiever is a *NOOP*. If we did then no cost would be added to account for the substitution. For example, consider the following situation where fact *(at {d1, d2} s1)* is reached by a *NOOP* with the sole precondition *(at d1 s1)*. If the goal is *(at d2 s1)* and this action has been selected as the cheapest achiever then we would add *(at d2 s1)* as a new goal – which is the same as the goal we are trying to achieve. In the second run the *NOOP* would be selected again, but because we have marked this substitution as *done* no extra cost will be incurred. Instead, whenever a *NOOP* is selected as an achiever for a goal, we always use the former option to make the substitution.

Example 3.5.4 *Consider the relaxed plan generated by Example 3.5.3. Here we have to make a substitution because the effect of the action* (walk d1 p1-3 s3) *does not achieve the goal* (at d2 s3)*, instead it achieves* (at d1 s3) *so we need to make a substitution between* $d2$ *and* $d1$*. The first approach searches for a fact layer where an equivalent object set is created that contains the annotated objects for* $d2$ *and* $d1$*. The earliest fact layer where this is the case is the second, so we add* 2 *to the total heuristic.*

If we use the second approach we add a new goal (at d2 p1-3) *to* G*. In order to solve this fact we find a* NOOP *from* (at {d1, d2} p1-3) *to* (at d1 p1-3)*. In this case, because it is a* NOOP*, we fall back on the former approach and add* 2 *to the heuristic estimate.*

3.5.5 Helpful actions

In addition to the heuristic FF uses pruning techniques to speed up search. Given a state s we can reduce the search space by only considering a subset of all the possible successor states. Ideally we want to restrict the successor states to those which lead to an optimal plan. However finding this subset is as hard as finding an optimal heuristic, which is as hard as solving the original problem. Instead, FF approximates this subset by using the solution of the relaxed planning problem.

Definition 45 — Helpful Action

Given a typed planning problem Π, the lifted relaxed planning graph rpg constructed from Π_{s_0}, and the extracted relaxed plan $\langle a_0, a_1, \ldots, a_n \rangle$ then we define helpful actions as a subset of all grounded actions. A grounded action $o \in \Pi_A$ is helpful if it has the following properties:

- o is applicable to Π_{s_0}.
- o achieves an effect that is present in the second fact layer rpg_{fl_1} and is used as a precondition for any action $a \in \langle a_0, a_1, \ldots, a_n \rangle$.

Helpful actions have been extended to all actions which are applicable to Π_{s_0} and are not restricted to those which are part of the relaxed plan. The reason for this is that the relaxed plan is constructed by ignoring delete effects, which means that the operators in the relaxed plan might not be helpful at all because they destroy previously

achieved goals or use too many resources. However, even by relaxing the set of actions that could be helpful, this method does not preserve completeness [32]. Despite the method not preserving completeness it has proved to be very successful on many domains using the h^{ff} heuristic.

We utilise helpful actions in the same way FF does. We prune the search space in such a way that we only consider helpful actions to generate the successor states. However, due to the way the lifted RPG is constructed and how relaxed plans are extracted this can distort the search we are conducting. For example, recall the relaxed plan used to find a plan to achieve *(driving d2 t1)* in Example 3.5.3:

- *(walk d1 s1 p1-3)*
- *(walk d1 p1-3 s3)*
- *(board d2 t1 s3)*

In this case any action which achieves *(at d1 p1-3)* (i.e. *(walk d1 s1 p1-3)*) is marked as helpful even though this action does not bring us closer to the goal. Due do the way we handle substitutions of equivalent objects we can construct helpful actions that do not actually help us to progress to the goal. While the helpful actions generated in this way will limit the size of the search space, it will direct the planner towards a suboptimal plan that will possibly include many unnecessary actions. More importantly, by executing these helpful actions we might not get a better heuristic estimate for the generated successor states.

This is a problem, because if we could prune the search space in such a way that we only consider helpful actions to generate successor states, then we would find the following solution to the problem:

- *(walk d1 s1 p1-3)*
- *(walk d1 p1-3 s3)*
- *(board d1 t1 s3)*
- *(walk d2 s2 p1-2)*
- *(walk d2 p1-2 s1)*
- *(walk d2 s1 p1-3)*
- *(walk d2 p1-3 s3)*
- *(disembark d1 t1 s3)*
- *(board d2 t1 s3)*

This is because the helpful actions are directed to get *d1* into the truck, because it requires less actions to get *d1* into the truck than making *d1* and *d2* equivalent. Eventually we reach the state where *d1* is driving the truck, which means that *d2* must get

inside the truck before it can become equivalent with *d1*, and this means that the helpful actions will direct *d2* to the goal state. The discrepancy between the identity of the objects in the helpful actions and those objects whose goal we are trying to accomplish will force the planner to execute many actions that do not contribute towards reaching the goal.

Therefore, we expect this pruning technique to yield better results when used in conjunction with making substitutions by adding new facts to the open list G instead of just adding a cost to the heuristic estimate. Returning to Example 3.5.3, when we try to find an achiever for *(at d2 s3)* we only have a single achiever which is *(walk d1 s1-3 s3)*. The variable domains of the precondition *(at d1 p1-3)* do not match up with the required precondition *(at d2 p1-3)*. Instead of finding the earliest fact layer where $d2$ and $d1$ are equivalent we add the new goal *(at d2 p1-3)* to G.

The fact *(at d2 p1-3)* has two achievers. The first is a *NOOP* with the precondition *(at d1 p1-3)*, the second is *(walk* $\{$*d1, d2*$\}$ *s1 p1-3)* with the precondition *(at* $\{$*d1, d2*$\}$ *p1-3)*. While the former has a cheaper action cost we select the latter because the former needs substitutions whilst the latter does not. By choosing the latter we find the relaxed plan $\langle$ *(walk d2 p1-2 s1)*, *(walk d2 s2 p1-2)* $\rangle$. This allows *(walk d2 p1-2 s1)* to be a helpful action which gives us the following relaxed plan:

- *(walk d2 s2 p1-2)*
- *(walk d2 p1-2 s1)*
- *(walk d2 s1 p1-3)*
- *(walk d2 p1-3 s3)*
- *(board d2 t1 s3)*

This plan leads to the optimal solution for this particular instance.

3.5.6 Preserve goals

h^{lrpg} relaxes the original problem by ignoring delete effects. This means that the relaxed plan will sometimes destroy goals that have been achieved or use resources that are no longer available. To generate better helpful actions and better heuristics estimates we try to preserve goals that have already been achieved. This is done by not allowing any operators that have an effect of removing any of the achieved goals. In addition we do not allow any object that appears on any of the goals to become equivalent with any other object. The last requirement is to prevent situations where objects that are equivalent in the initial state allow actions to be executed that delete established goals.

By restricting the construction of the lifted RPG so that no achieved goals can be removed we generate better heuristic estimates. It is possible, however, that imposing these constraints makes it impossible to find a relaxed plan. In that case it is necessary that achieved goals must be violated in order to achieve other goals. In this case we reconstruct the lifted RPG and allow actions which violate goals that have already been achieved in order to find a relaxed plan.

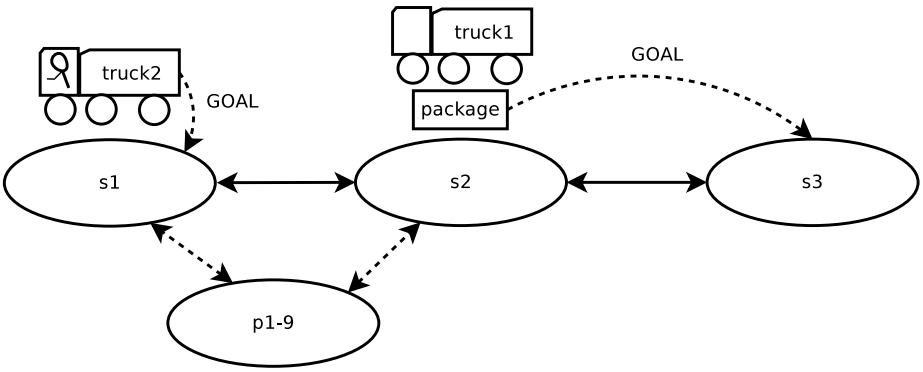

Figure 3.10: Example *Driverlog* domain with two goals: *(at truck2 s1)* and *(at package s3)*.

Example 3.5.5 *Consider the* Driverlog *problem depicted in Figure 3.10. This planning problem has two goals that need to be achieved:* (at truck2 s1) *and* (at package s3). *As we can see, one of the goals has already been achieved. However, if we construct a lifted RPG and extract a relaxed plan we would find the following sequence:*

$$\langle (drive\ driver\ truck2\ s1\ s2),\\ (load\ package\ truck2\ s2),\\ (drive\ driver\ truck2\ s2\ s3),\\ (unload\ package\ truck2\ s3)\rangle$$

However, if we restrict the construction of the lifted RPG so that the achieved goal (at truck2 s1) *cannot be violated we end up with the following sequence of actions:*

$$\langle (disembark\ driver\ truck2\ s1),\\ (walk\ driver\ s1\ p1-2),\\ (walk\ driver\ p1-2\ s2),\\ (board\ driver\ truck1\ s2)\\ (load\ package\ truck1\ s2),\\ (drive\ driver\ truck1\ s2\ s3),\\ (unload\ package\ truck1\ s3)\rangle$$

This is a better cost estimate than the previous estimate.

Preserving goals is a good technique to get better heuristic estimates (as we will see in the next section), but it does not always provide better estimates. In some cases it is beneficial to violate some goals temporarily. For example, imagine that the graph depicted in Figure 3.10 contains six more locations which are daisy chained to location *s3* as depicted in Figure 3.11 then it makes more sense to use *truck2* to deliver the package before driving it back to its location.

Fast Forward uses *goal ordering* to determine in which order goals should be achieved, which circumvents the above problem. We have not implemented this in our planner, but if we did we could impose constraints based on the sets of goals that

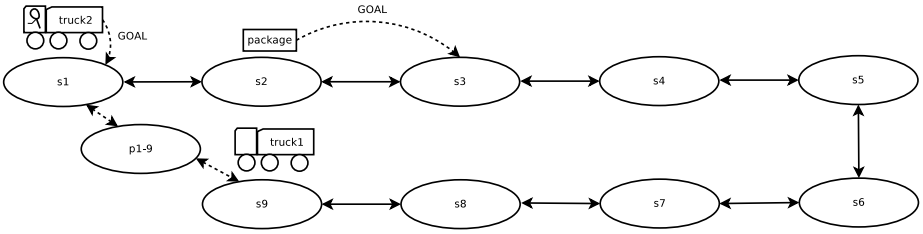

Figure 3.11: Example *Driverlog* domain with two goals: *(at truck2 s1)* and *(at package s3)*.

have been achieved. For example, if goal orderings were such that packages must be delivered first, followed by driving trucks to their locations and finally the drivers must get to their locations, then we could prevent only any packages that have reached their goals to be violated. We will return to this subject when we discuss future work.

3.6 Implementation of the planning system

Now that we have described how we construct the lifted RPG and presented multiple ways to extract a lifted heuristic we will now present the details of the planning system. As we have described in Section 2.6, heuristics – even when nearly perfect – are not sufficient to find a solution for some planning problems in a reasonable amount of time. Given that we have developed a *lifted* algorithm we cannot expect our heuristic to be as informative as h^{ff}. In order to solve larger problem instances we use the same incomplete algorithm as FF *enforced hill-climbing* and only use *helpful actions* to expand a state.

Given a planning problem $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ we create an open list G that initially contains the initial state s_0. While we have not found a solution to the problem and G is not empty, we pick a state from G with the *lowest* heuristic estimate and create all the successor states. If multiple states have the same heuristic estimate we pick one at random to expand. There is no guarantee, given a state s with heuristic value i and s' with heuristic value $i - 1$, that s' is closer to the goal than s. So we could be led astray based on the states we choose to expand.

Given a state s with heuristic value i, enforced hill-climbing commits to the first state it finds with a heuristic value $j < i$. If we do not find a better state after expanding a certain number of states we restart search from the last state we committed to, which includes the initial state. After every restart we double the amount of states allowed to be extended before we restart. After a certain number of restarts we fall back on *best-first search* and do not use any pruning techniques such as helpful actions. The algorithm is listed in Algorithm 5, the value of the parameters $ignore\_unhelpful$ and $preserve\_goals$ depend on the configuration used (see Table 3.3);

3.7 Results

Now that we have discussed the planning algorithm and the lifted heuristic we will now run experiments to test our hypothesis that our planner uses less memory than state-of-the-art planners that use grounding. In addition we want to test if our planner scales better and is able to solve problems that grounded planners cannot solve due to memory constraints. We use the FF planner for comparison, because its search algorithm and heuristic are closest to our planning system. In addition to comparing memory usage we also check the plan quality and the number of states that have been expanded to solve each problem to confirm our hypothesis that our heuristic is informative despite being lifted.

We have implemented all ideas discussed above and will now show how our planner using h^{lrpg} compares to FF using h^{ff}. The configurations we compare with FF are listed in Table 3.3.

We compare each configuration against the Fast Forward heuristic h^{ff}. We compare the time taken to solve a problem, states visited, plan quality, and memory used.

We ran all our experiments on an Intel Core i7-2600 running at 2.4GHz and allowed 2GB of RAM and 30 minutes of computation time. We have taken seven problem domains from various planning competitions, and we now discuss the results obtained in these domains.

3.7.1 Naive

The first configuration does not use any substitutions nor any pruning techniques. This configuration serves as the baseline to test the benefits of the two substitution methods in combination with helpful actions and *preserve goals*. We expect that this configuration is able to explore a larger part of the search space because it is able to calculate heuristic estimates faster, but due to the poor quality of these heuristics we do not expect this configuration to be as good as the other configurations.

We compare the number of states explored versus the FF heuristic in Figure 3.12. We can see that our algorithm is less informative than FF which is what we expected. The plan quality is depicted in Figure 3.13 and is very poor. This too is in line with our expectations, because the heuristic guidance is worse than the FF heuristic. Finally we compare the time it takes to solve the planning problems, the results are depicted in Figure 3.14. It shows that we are much slower than FF, this is because a number of reasons. First of all we explore more states than FF, but this is not sufficient to explain the difference in time which is two orders of magnitude slower. In addition FF uses helpful actions to aggressively prune the search space while we calculate the heuristic value of all successor states and do not prune at all. The subsequent configurations improve upon this naive implementation.

3.7.2 Enhanced Fully Lifted

The second configuration uses the ObjectSub substitution method. The heuristic estimates it calculates is a massive improvement over the baseline and the overhead of resolving substitutions is very small so we expect that this configuration will solve many

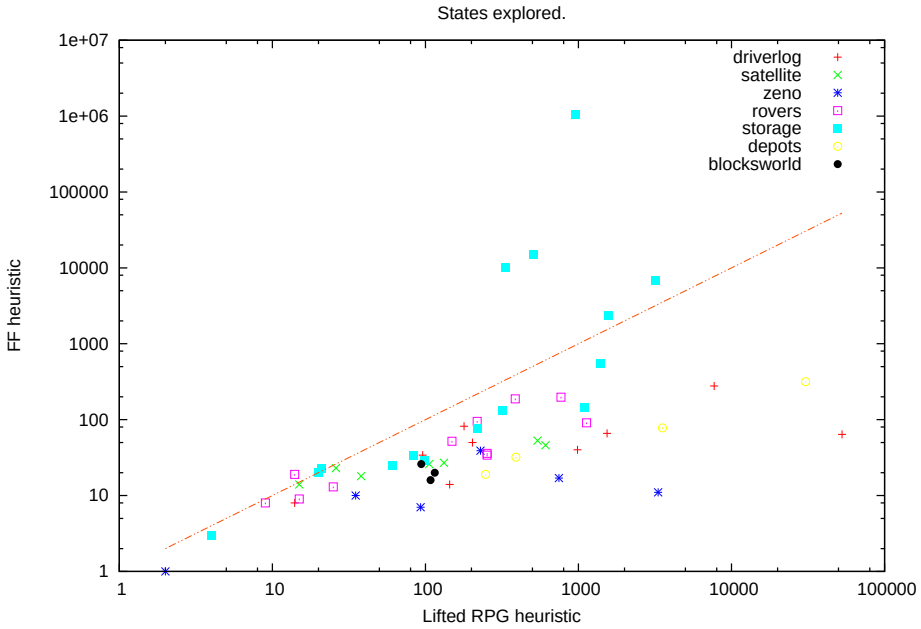

Figure 3.12: Naive lifted RPG heuristic.

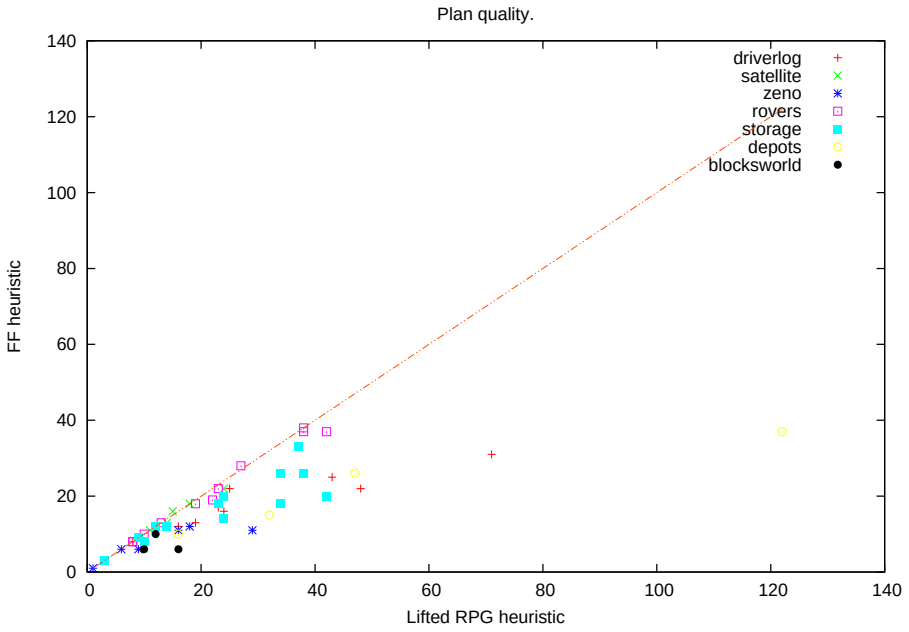

Figure 3.13: Naive lifted RPG heuristic.

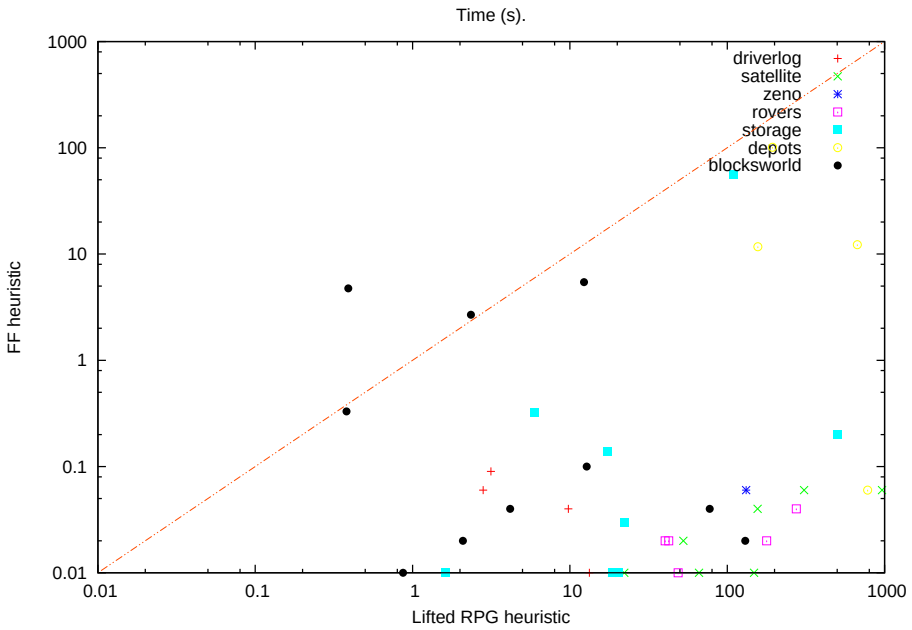

Figure 3.14: Naive lifted RPG heuristic.

more problems. We also ran experiments where we used helpful actions to prune the search space. However, as we have explained in the previous section, helpful actions do not always help the planner and can misdirect it. We have include this configuration to test this hypothesis. The final configuration uses helpful actions to prune the search space and it constrains the construction of the lifted RPG by trying to preserve goals. We hope that the helpful actions extracted from the constrained lifted RPG will better guide the planner to a goal state.

The results for the configuration where we do not prune unhelpful actions and do not preserve goals are depicted in Figure 3.15 for the number of states explored and Figure 3.16 depicts the plan quality. We only depict points for problems solved by both planners. As we can see, handling substitutions improves the number of problems we can solve significantly and improves the informativeness of the heuristic because we need less states to explore before we find a solution.

Interestingly, pruning unhelpful action actually decreases the performance as we can see from Figure 3.17 and Figure 3.18. While some problems are solved quicker, most of the problems take longer to solve and we solve less problems overall. Recall the discussion from Section 3.5.5 that pruning unhelpful actions can distort the search if handle substitutions between any two objects o and o' by finding the minimum layer number where o and o' are equivalent.

Results for the third configuration (where we prune unhelpful actions and try to preserve goals) are depicted in Figure 3.19, Figure 3.20 and Figure 3.21. As we can see, preserving goals proves to be a very successful pruning method and decreases the

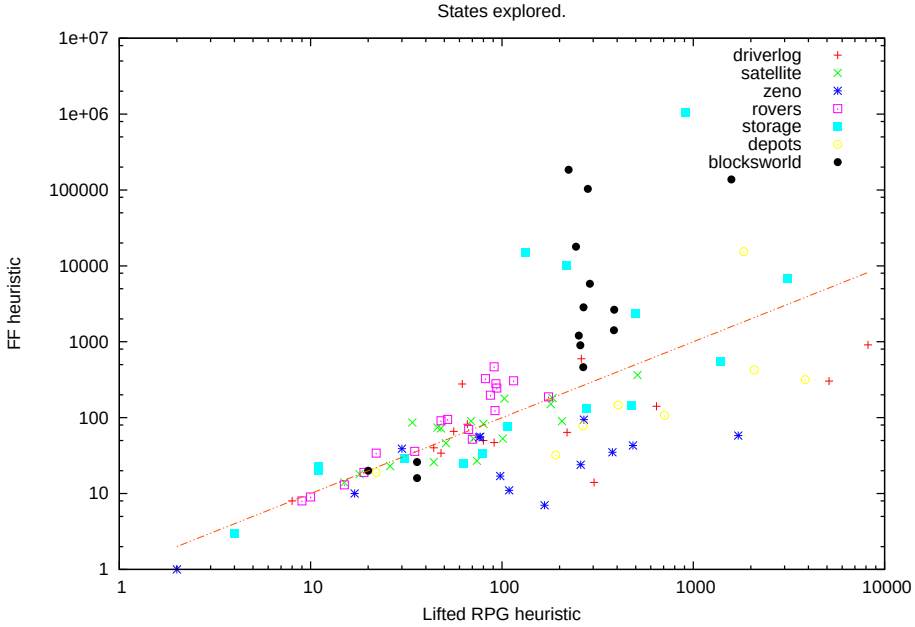

Figure 3.15: Enhanced fully lifted RPG heuristic, do not preserve goals.

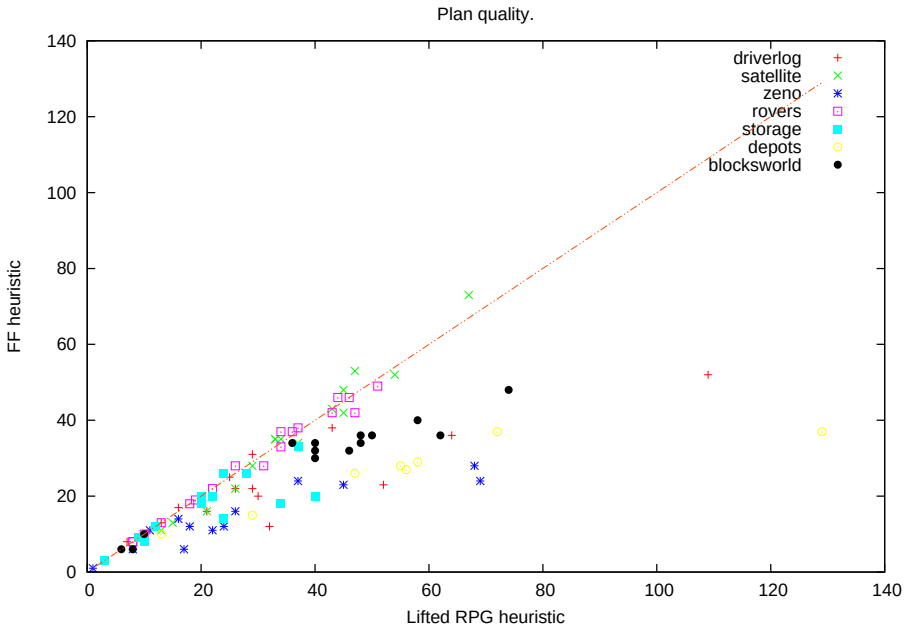

Figure 3.16: Enhanced fully lifted RPG heuristic, do not preserve goals.

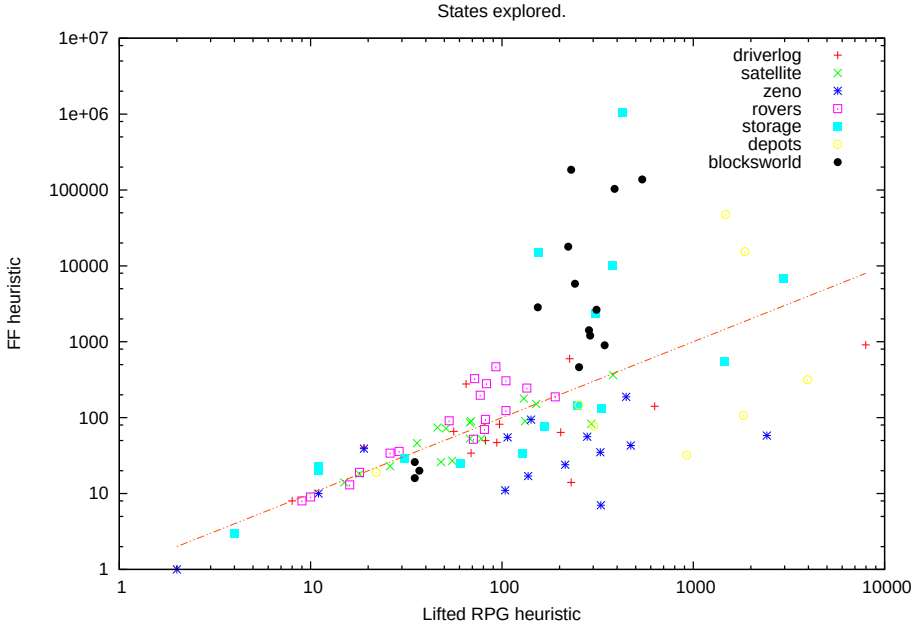

Figure 3.17: Enhanced fully lifted RPG heuristic, prune unhelpful actions.

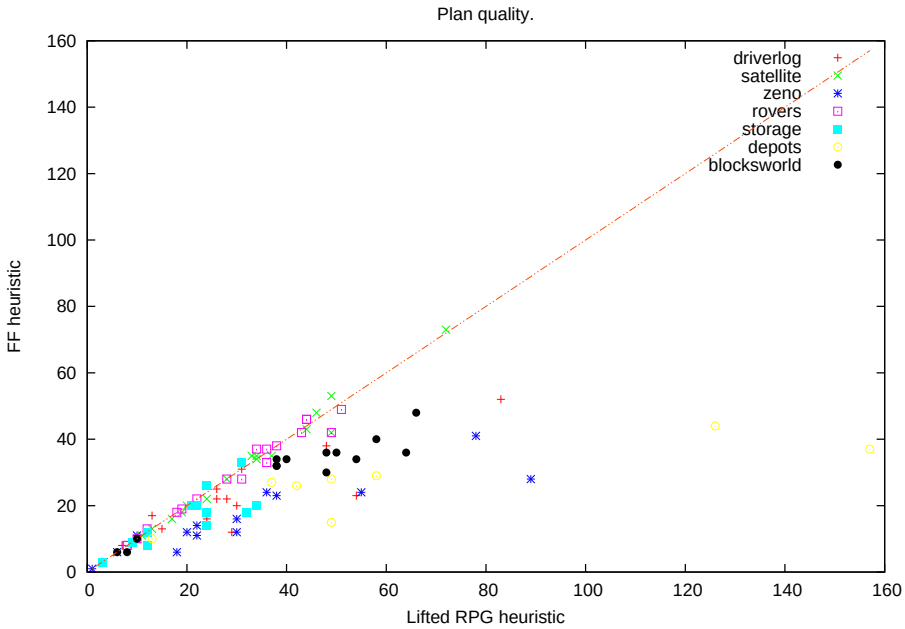

Figure 3.18: Enhanced fully lifted RPG heuristic, prune unhelpful actions.

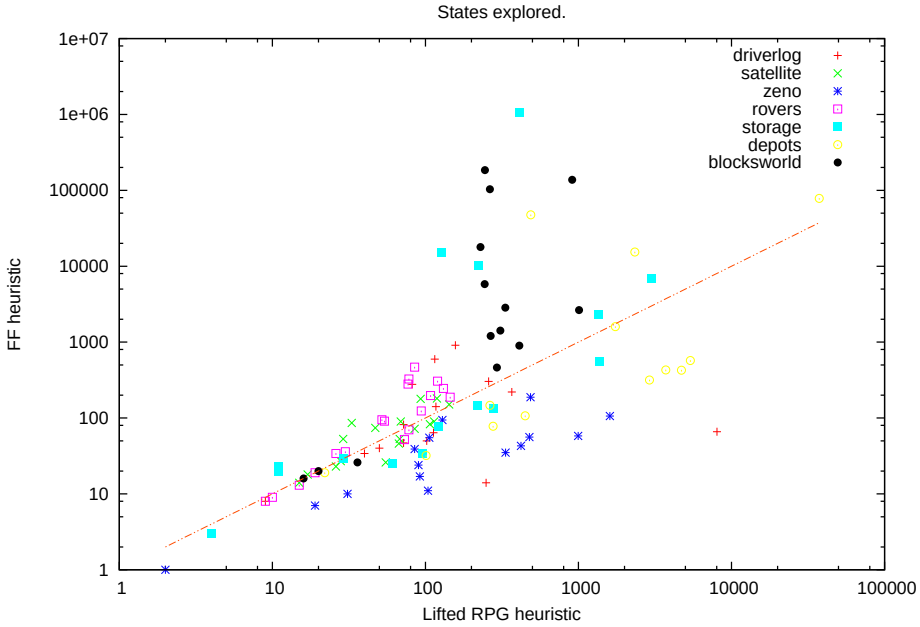

Figure 3.19: Enhanced fully lifted RPG heuristic, all features enabled.

number of states we need to visit and allows us to find solutions for more problems.

Surprisingly our lifted algorithm explores less states to find a solution for domains like *Blocksworld*, *Storage*, and *Rovers*. The results for *Zeno* and *Depots* favour FF while the results for the other domains are comparable. When we compare the time we observe that we take less time than the naive implementation but because we calculate the heuristic value of all helpful actions we are still slower than FF. FF commits to the first successor which has a better heuristic value than the parent node and is must faster for that reason. In addition while *reachability analysis* is relatively fast (see Section 3.4) the extraction of a relaxed plan is slower given a lifted RPG due to resolving substitutions.

When we compare for plan quality it is clear that FF produces shorter plans. The reason for this has to do with the abstractions we use; recall Example 3.5.3 where the relaxed plan includes actions to move driver $d1$ towards the truck whereas $d2$ is the object we want to board the truck. We try to detect these discrepancies by adding new goals to the set of goals G, but this does not always help. In some situations we find that moving a different object towards the goal decreases the heuristic estimate because it introduces new shortcuts due to objects being made equivalent in earlier fact layers.

For example the problem in Figure 3.22 is to get $d1$ to location G. The length of each dotted path is n. The problem with this structure is that $d2$ will be made equivalent with $d1$ at fact layer $2n + 1$, but *(at d2 G)* is made true at fact layer $2n$. Thus, when extracting a relaxed plan we find a plan of length $2n$, but then we need to substitute $d1$ for $d2$ which will cost $2n + 1$ for a total heuristic value of $4n + 1$. By contrast, FF

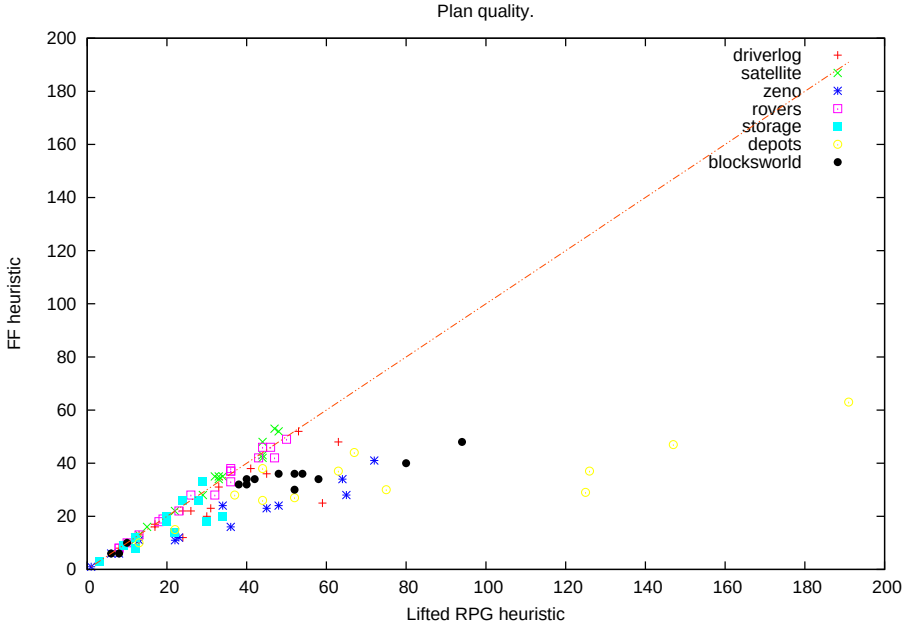

Figure 3.20: Enhanced fully lifted RPG heuristic, all features enabled.

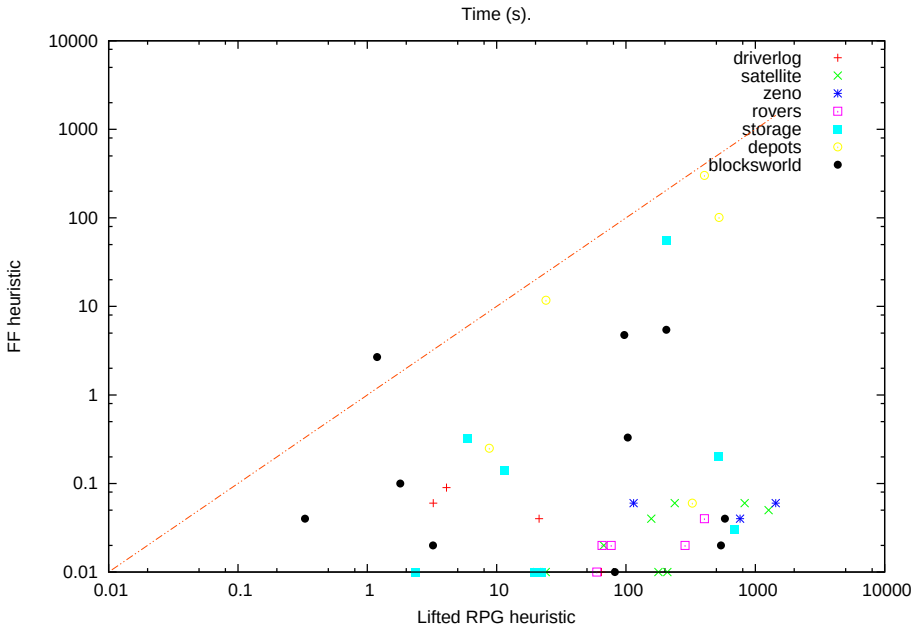

Figure 3.21: Enhanced fully lifted RPG heuristic, all features enabled.

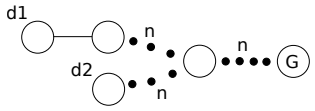

Figure 3.22: A case where our heuristic performs poorly.

will find the correct heuristic value of $2n + 1$. In this case moving $d1$ and $d2$ will both produce a better heuristic value.

Even though we handle substitutions by adding new goals to the relaxed plan, there are situations where we end up in similar situations where moving an object that is not relevant to the goal introduces shortcuts which affects the heuristic value.

3.7.3 Enhanced Partially Lifted

The third and last configuration uses the second substitution method. This configuration calculates the best heuristic estimates. However, the overhead to calculate these heuristic estimates is the highest of all configurations because it is possible that many new goals are added to the lifted RPG before we find a heuristic estimate. Like the *ObjectSub* configuration we test the effectiveness of pruning the search space with helpful actions and test if constraining the lifted RPG by preserving goals helps the quality of the heuristic estimate and the helpful actions. An additional benefit of constraining the lifted RPG by preserving goals is that it limits the number of substitutions we need to make which will improve the speed of calculating heuristic estimates.

The first configuration we show is one where we do not use any helpful actions and we do not try to preserve goals that have already been achieved. The plan quality of the found solutions is depicted in Figure 3.24, we see that our solutions still suffer from being led astray due to the substitutions that need to be made, but less so than the *Object Sub* configuration. The number of states that are explored is depicted in Figure 3.24. We see that our approach performs a lot worse than FF. This is not unexpected because we do not use helpful actions and other pruning actions FF has at its disposal.

Handling substitutions by updating the structure of the lifted RPG is a worse strategy compared to handling substitutions by augmenting the heuristic estimate with the layer number where objects become equivalent. However, when we enable helpful actions we get a very different picture. The results for this configuration are depicted in Figure 3.25 the the number of states explored and Figure 3.26 for the quality of the found plans. The reason is that helpful actions no longer lead the search astray as it did with the *enhanced fully lifted* configuration.

The last configuration tries to preserves goals whilst still pruning unhelpful actions. The results for this configuration are depicted in Figure 3.27 for the number of states explored, Figure 3.28 for the quality of the found plans and Figure 3.29 for the time needed to find a solution. We observe that we have to expand considerably fewer states than does FF for the *Rovers*, *Blocksworld*, *Satellite*, *Driverlog*, *Storage*, and *Depots* domains. The only domain where we need to expand many more states is the *Zeno* domain. When we compare the plan quality then we see that FF produces plans of better quality. Even though we produce better quality plans than the fully

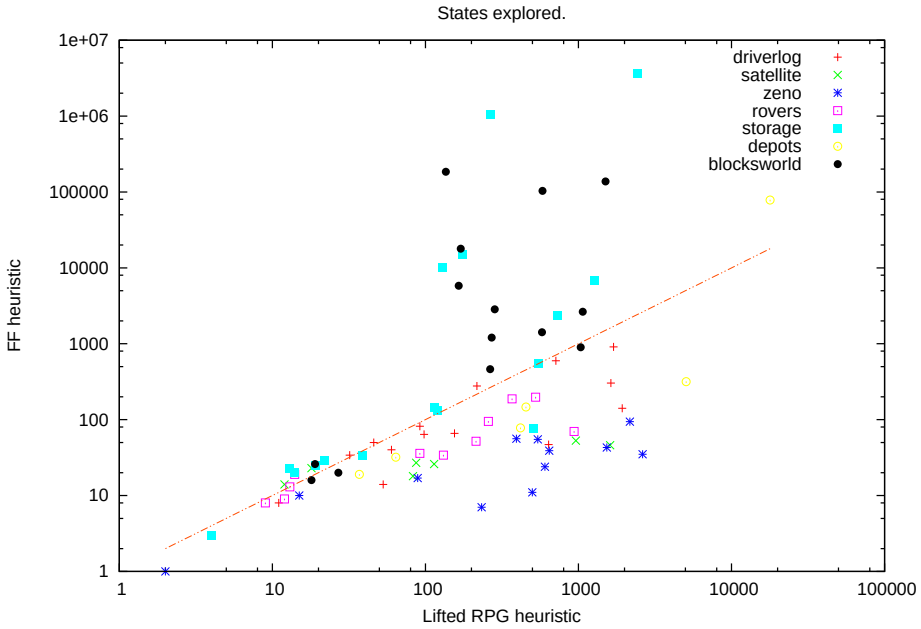

Figure 3.23: Enhanced partially lifted RPG heuristic, no features enabled.

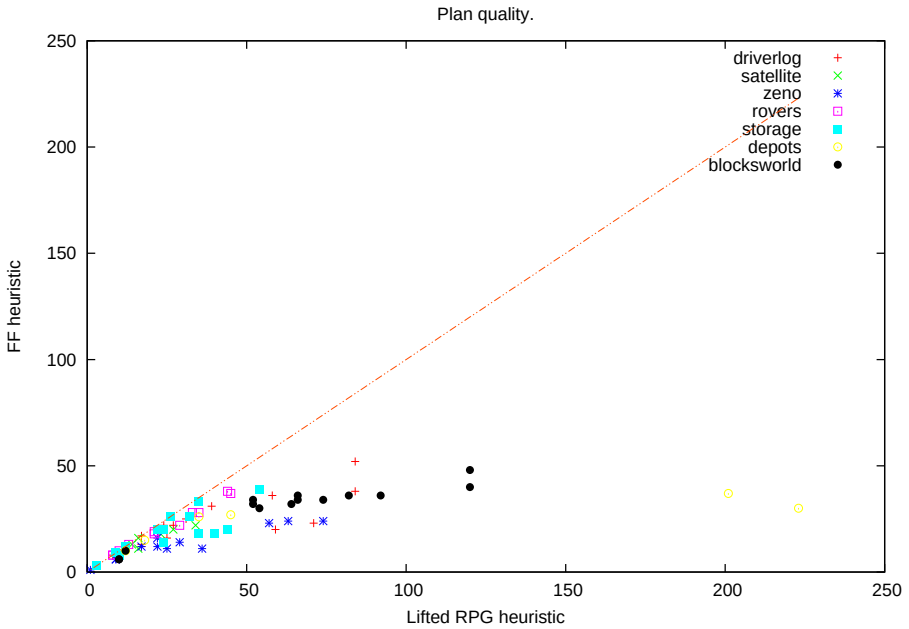

Figure 3.24: Enhanced partially lifted RPG heuristic, no features enabled.

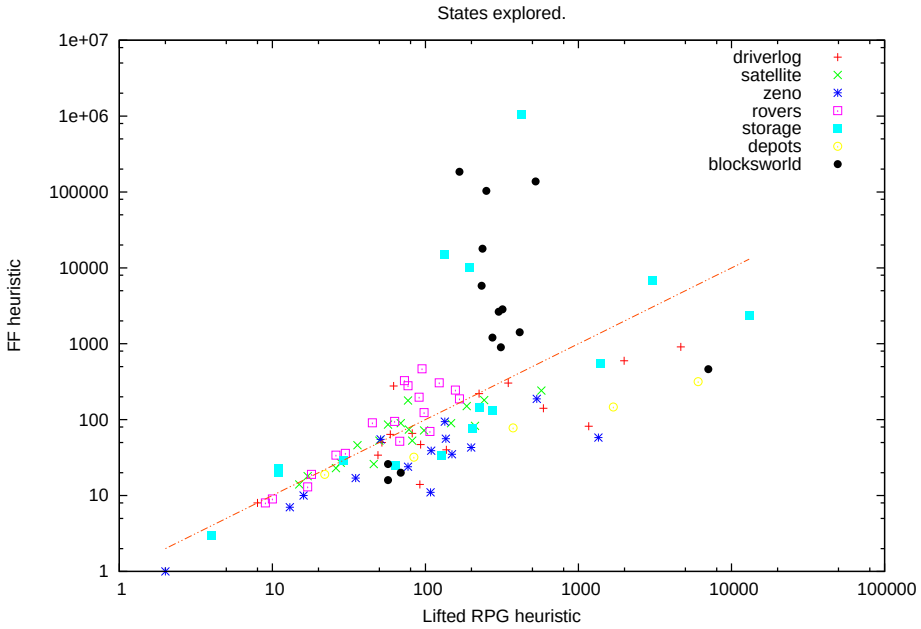

Figure 3.25: Enhanced partially lifted RPG heuristic, prune unhelpful actions.

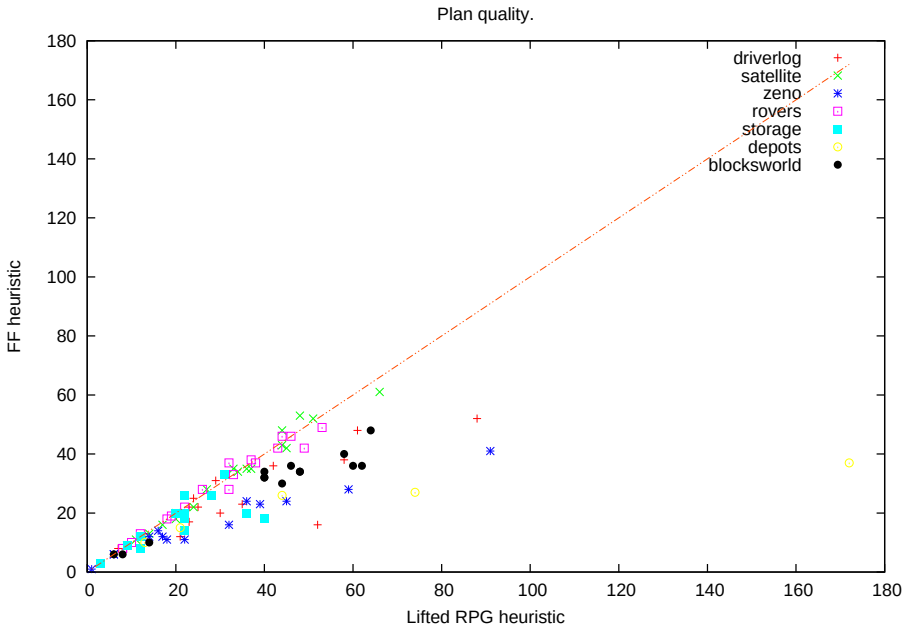

Figure 3.26: Enhanced partially lifted RPG heuristic, prune unhelpful actions.

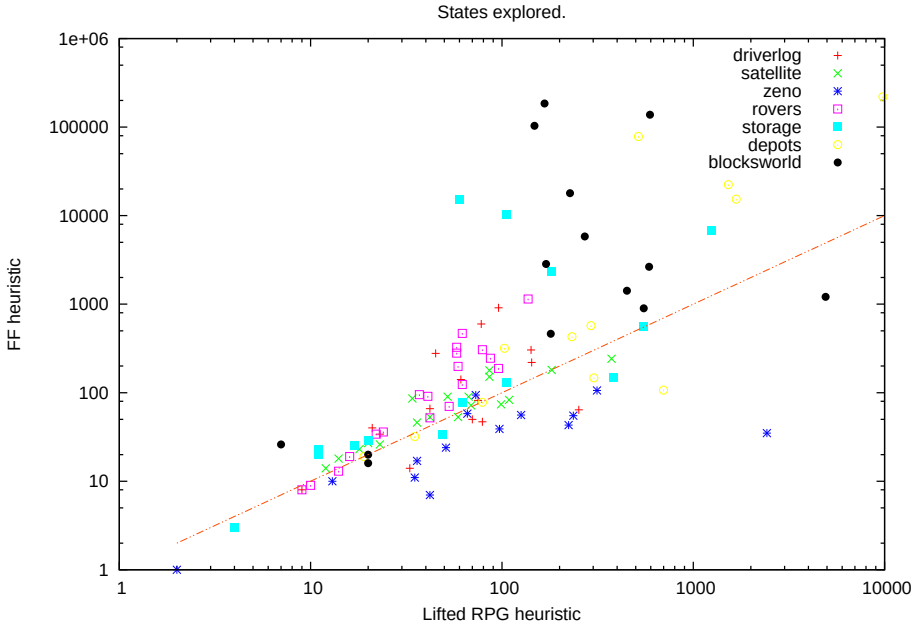

Figure 3.27: Enhanced partially lifted RPG heuristic, all features enabled.

lifted configuration we still suffer from having to make substitutions. We suffer less because we can detect more instances when we need to make substitutions than does the fully lifted configuration. When we compare the time it takes we do not see a huge improvement over the *enhanced fully lifted* configuration. While we explore less states it takes us longer to evaluate these states because the number of actions in the relaxed plan are not bounded to the number of actions in the lifted relaxed planning graph.

3.7.4 Summary

The number of planning problems solved for each configuration are listed in Table 3.4. We can see that the naive implementation of our planner – which does no pruning and does not make any substitutions – performs worst of all. This is as we expected, because the heuristic used is not as informative as the FF heuristic. We see that the performance almost doubles when we enable making substitutions to account for the dependencies between the variable domains of the actions in the relaxed plan and the preconditions in the lifted RPG.

We see that the performance improves even more when we enable helpful actions and constrain the construction of the lifted RPG so that no achieved goals are deleted. However, we are not able to match the performance of FF. This is because our heuristic estimates are not as good as those of FF. Even though we have introduced novel pruning techniques, there are a few techniques incorporated in FF that are not part of our planner. Most notable is the goal ordering that FF utilises; these techniques are

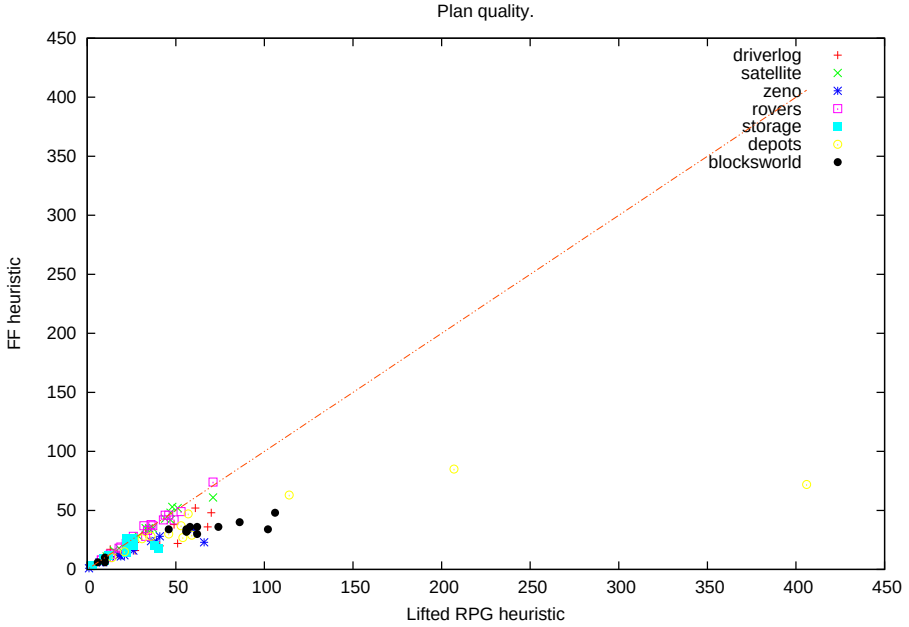

Figure 3.28: Enhanced partially lifted RPG heuristic, all features enabled.

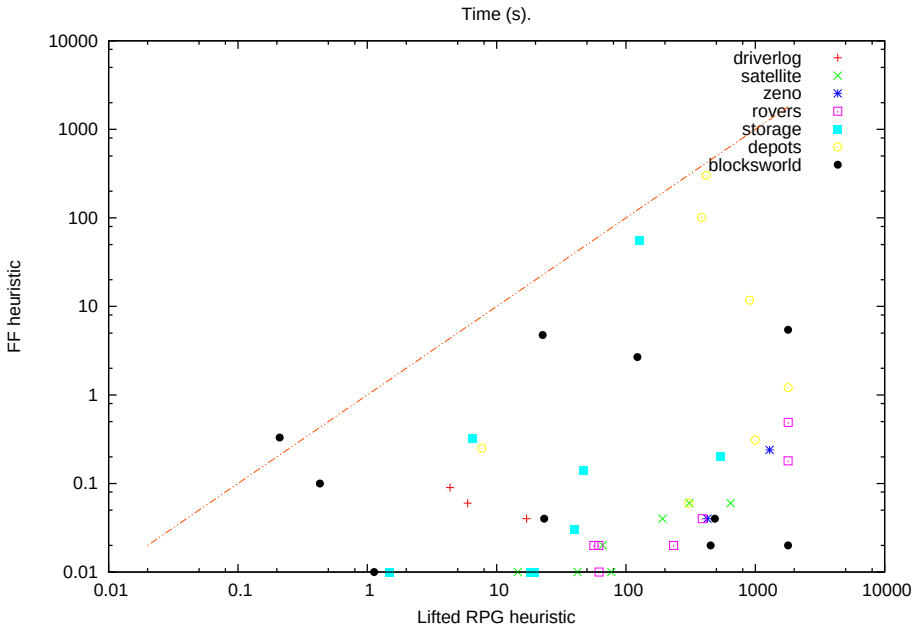

Figure 3.29: Enhanced partially lifted RPG heuristic, all features enabled.

applicable to our planner but are not presented here and left for the discussion on future work. Further, we do not prune as aggressively as FF. When FF processes a state s and generates its successors it stops whenever it finds a state with a better heuristic value than s, whereas we generate *all* successors and commit once a state is selected to be expanded. We have implemented a version which prunes more aggressively and we will present those results later. Lastly, while we construct the lifted RPG more quickly than planners can construct an RPG (on domains which exhibit a lot of *almost symmetry*) it can take longer to extract a relaxed plan from a lifted RPG if we allow partial grounding and make substitutions by adding new goals to the lifted RPG. This is the reason why we see little improvement between the *enhanced fully lifted* setup and the *enhanced partially lifted*, because while the latter does provide better heuristic estimates and better helpful actions it takes longer to compute, so fewer states can be expanded.

3.7.5 Memory results

To verify that our partially lifted RPG heuristic uses less memory than the FF heuristic we recorded memory usage for all instances of each configuration used in our tests. The results for the naive configuration are depicted in Figure 3.30. We use significantly less memory than FF to solve the problems, although we seem to scale worse than FF for certain domains, most notably Rovers and Storage. Note that we only depict results for problems that both planners solved. The reason why we scale relatively poor on the Rovers and Storage domains is because there are few objects that can be made equivalent.

The results for the enhanced fully lifted configuration are depicted in Figure 3.31. We see more data, because we solve more planning problems. We see that a few of the Blocksworld planning problems seem to scale badly. On these problems we reached plateaus and had to store a great number of states before we could escape the plateaus or discover that we could not found a solution by pruning actions which are not helpful.

The results depicted in Figure 3.32 show that out most memory intensive configuration still consumes less memory than FF. This confirms our claim that while this configuration could in theory ground all the actions in order to construct the lifted relaxed planning graph, in most cases this is not the case. Also note that we do not have any outlines for the Blocksworld domain depicted in Figure 3.31 because the heuristic estimate is more accurate. This configuration finds a solution quicker and subsequently can solve more problem instances as depicted in Table 3.4.

We could further reduce our memory overhead by pruning more aggressively like FF does. While FF generates the successor states S for a state s it selects the first state $s' \in S$ for which $h^{ff}(s') < h^{ff}(s)$. We generate and store all successor states which increases our memory usages. Our heuristic is not as informative as the FF heuristic which means that we need to store and explore more states in order to escape from plateaus. We can reduce the memory significantly by applying more aggressive pruning techniques.

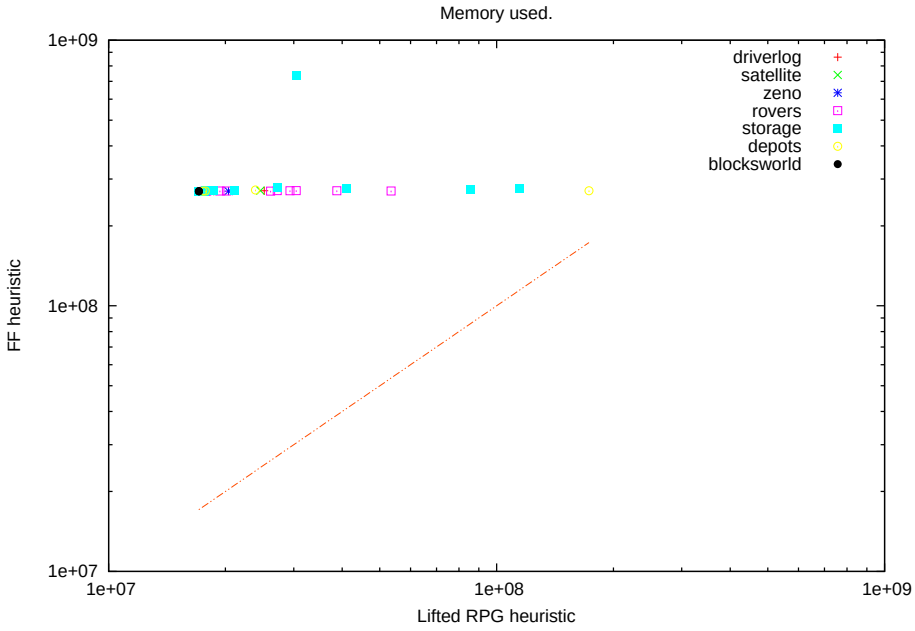

Figure 3.30: Naive RPG heuristic.

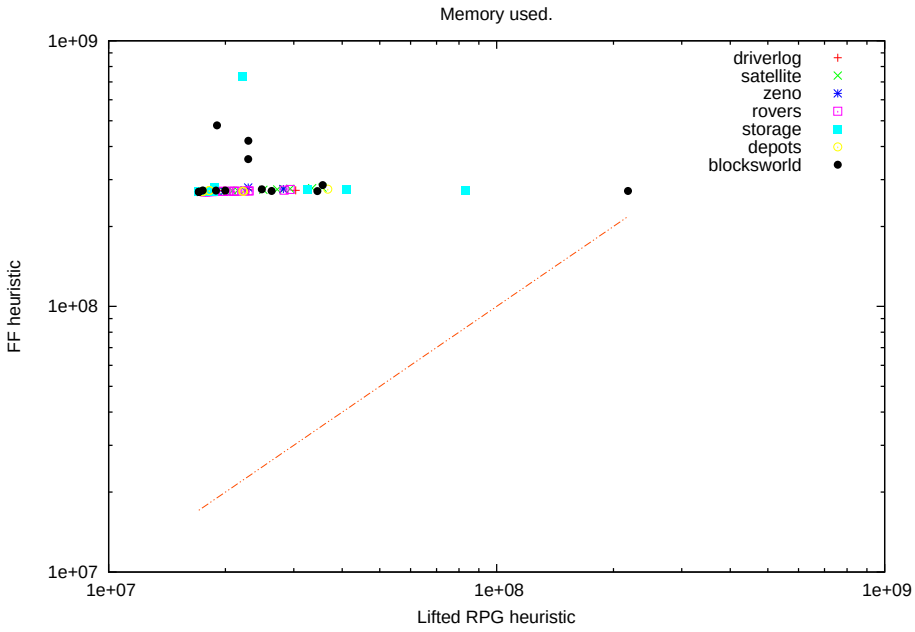

Figure 3.31: Enhanced fully lifted RPG heuristic, all features enabled.

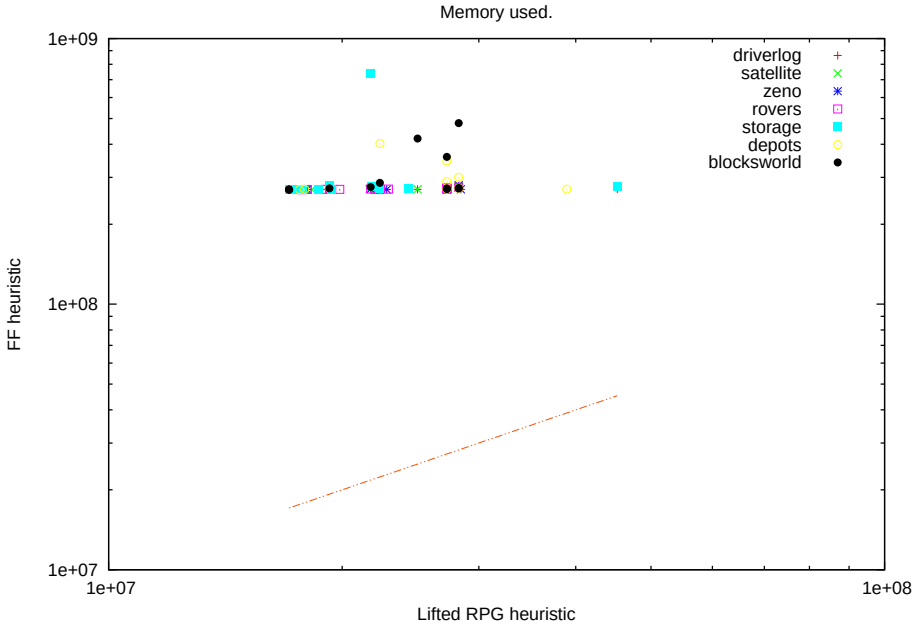

Figure 3.32: Enhanced partially lifted RPG heuristic, all features enabled.

3.7.6 Larger domains

To test our hypothesis that our planner can solve larger problem instances we have created a simple domain called *Pigeonhole*. We have already demonstrated that our planner can work on bigger problem instances (see Section 3.4). In this section we will show problem instances we can solve but FF, due to memory constraints, cannot. The pigeon domain is a simple one: the initial state contains n *(flying pigeon)* predicates, there is a single action *(fly pigeon hole)*, and the goal consists of n *(in pigeon hole)* facts. The number of actions in a domain is equal to n^2. We ran an experiment to see how much memory is required to solve this problem instance. The results are depicted in Figure 3.33.

We have used the *enhanced partially lifted* configuration with pruning helpful actions and preserving goals. In order to solve this problem we cannot use the *naive* or *enhanced fully lifted* configuration. The reason is that both these configurations would return a relaxed plan which consists of a single action:

$$(fly\{pigeon0, pigeon1, \ldots, pigeonn\}\{hole0, hole1, \ldots, holen\})$$

So there is no heuristic guidance and the set of helpful actions is the entire set of grounded actions. So this method would be as bad as the original FF heuristic and be slower since there is no benefit to offset the overhead we incur. We tested domains up to 1,600 pigeons and holes, simply because FF cannot solve this instance due to memory constraints and *VAL* [34] cannot deal with bigger problems. Given these results we

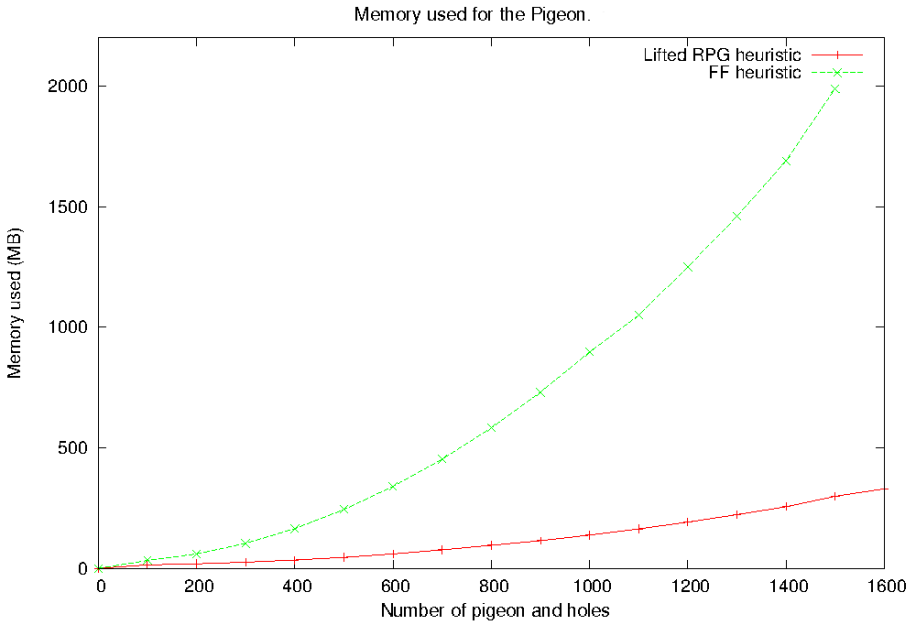

Figure 3.33: Comparison of the memory used by FF and our planner.

expect we can solve problem instances up to $10,000$ pigeons, which we consider to be a very large number because grounding such a domain would mean having $100,000,000$ grounded actions, more than any state-of-the-art planner can handle at the moment.

Algorithm 5: The planning algorithm.

```
findPlan(ignore_unhelpful, preserve_goals)
P = search(ignore_unhelpful, preserve_goals, TRUE);
if P = ∅ P = search(FALSE, preserve_goals, FALSE);
return P;
search(ignore_unhelpful, preserve_goals, enable_ehc)
h_lowest = ∞;
priority_queue = {⟨s_0, {}, ∞⟩};
closed_list = {};
states_before_restart = 1;
states_explored = 0;
last_best_state = s_0;
last_best_plan = {};
while priority_queue ≠ ∅ do
    ⟨s, P, h⟩ ∈ priority_queue | ¬∃⟨s', P', h'⟩ ∈ priority_queueh' < h;
    priority_queue = priority_queue \ ⟨s, P⟩;
    if s ∈ closed_list then
        continue;
    closed_list = closed_list ∪ s;
    if s_g ⊆ s then
        return P;
    for ⟨s', a⟩ ∈ successorsof s do
        if ignore_unhelpful AND a is not an helpful action then
            continue;
        h_s = ∞;
        if preserve_goals then
            lrpg_preserve = lifted relaxed planning graph constructed for s
            preserving all atoms in s_g;
            h_s =| the relaxed plan extracted from lrpg_preserve;
        if heuristic_value ≡ ∞ then
            lrpg_nopreserve = lifted relaxed planning graph constructed for s;
            h_s =| the relaxed plan extracted from lrpg_nopreserve;
        if h_s ≡ ∞ then
            continue;
        states_explored = states_explored + 1;
        if h < h_lowest then
            states_explored = 0;
            last_best_state = s';
            last_best_plan = {P ∪ a};
            if enable_ehc then
                h_lowest = h;
                priority_queue = ∅;
        if enable_ehc AND states_explored ≡ states_before_restart then
            priority_queue = {⟨last_best_state, last_best_plan⟩};
            states_before_restart = states_before_restart * 2;
        else
            priority_queue = priority_queue ∪ {⟨s', {P ∪ a}, h_s⟩};
return ∅;
```

Configuration Name	prune unhelpful actions	preserve goals	substitution method
Naive	No	No	None
Enhanced Fully Lifted	No	No	ObjectSub
Enhanced Fully Lifted (h)	Yes	No	ObjectSub
Enhanced Fully Lifted (h+p)	Yes	Yes	ObjectSub
Enhanced Partially Lifted	No	No	GoalSub
Enhanced Partially Lifted (h)	Yes	No	GoalSub
Enhanced Partially Lifted (h+p)	Yes	Yes	GoalSub

Table 3.3: The different configurations of our planning system.

	NoSub	ObjectSub			GoalSub			
Domain	default	default	h	h+p	default	h	h+p	FF
Driverlog	9	14	13	15	14	15	18	15
Zeno	6	13	13	15	12	14	14	20
Blocksworld	3	20	20	20	20	20	20	14
Storage	17	17	17	17	18	17	16	18
Depots	4	8	8	13	6	9	12	20
Satellite	7	18	17	17	7	18	18	20
Rovers	11	18	18	18	13	18	19	20
Total	57	108	106	115	90	110	117	127

Table 3.4: Number of problems solved. h means helpful actions enabled. p means that goals are preserved.

Chapter 4

Lifted causal graph heuristic

In this section we will show how we have adapted the h^{cg} heuristic without having to ground all the actions, as we have done with the Fast Forward heuristic. We will describe all the techniques and methods used to compute the *lifted causal graph* heuristic, or h^{lcg}. We will compare our approach to the causal graph heuristic h^{cg}, the context enhanced additive heuristic h^{cea}, and the merge and shrink heuristic h^{ms}. These algorithms are chosen for comparison because they are closest to our proposed *lifted causal graph* heuristic.

This section is ordered as follows. In the first section we explore how we exploit the property and attribute spaces constructed by TIM to construct data structures used to calculate h^{lcg}. These data structures are reminiscent of the *domain transition graphs* constructed by Fast Downward. An important difference with the construction of DTGs is that we do not need to fully ground the domain. Next we reduce the dependencies between state variables by merging *lifted transition graphs*. The method we use to merge is similar to the method used by *Merge and Shrink*. Next we explain how we use these structures to calculate the lifted causal graph heuristic. Finally we compare the lifted causal graph heuristic with related heuristics in the results section.

4.1 Knowledge compilation using TIM

As has been observed before [9, 26] and as we have discussed in Section 2.4.2, encoding a state as a set of literals that can be made *true* or *false* is inefficient. Many planning problems contain cliques of atoms that are mutually exclusive. These cliques allow for a more concise state representation where a state no longer consists of a set of literals which can either be *true* or *false*. Instead we create a *state variable* for each mutually exclusive clique whose value domain is equal to the set of atoms in the clique. Given a set of state variables, we can represent a state by assigning a single *value* to each *state variable*.

Techniques such as TIM [15], DISCOPLAN [21], and SAS+ translation techniques [27] find state variables and allow a typed planning problem to be transformed into a multi-valued typed planning problem where mutually exclusive cliques of atoms

are transformed into state variables that can take a set of values (see Section 2.6.1).

We use TIM to extract state variables from a typed planning problem. We choose TIM because TIM extracts this information without having to ground the planning problem. Recall from Section 2.6.1 that TIM finds cliques of *sets* of atoms that are mutually exclusive. We redefined the definition of a typed multi-valued planning problem to account for the fact that the value of a state variable can be a *set* of atoms instead of a single atom, because the preconditions or effects of an action does not have to include all the atoms of a value of a state variable.

Definition 46 — Typed Multi-Valued Planning Problem
A typed multi-valued planning task is a tuple $\Pi = \langle T, O, P, Q, A, s_0, s_g \rangle$ where:

- T is a set of types. Every type $t \in T$ has a set of supertypes, written $SuperType(t)$.
- O is a set of objects. Each object $o \in O$ is associated with a type $t \in T$, written $Type(o)$.
- P is a set of predicates, where a predicate $p \in P$ is a tuple $\langle name, types \rangle$.
- Q is a finite set of state variables, each with an associated finite domain D_q. Each element in D_q is set of atoms. An atom is a tuple $\langle p, V \rangle$, where $p \in P$ and V is a set of variables. A variable v is a pair $\langle t, D_v \rangle$, where $t \in T$ and $D_v \subseteq O$ is the domain. We refer to the ith variable with the notation V_i. If the size of all domains of all variables of an atom is exactly one we call that atom *grounded*. A partial variable assignment or partial state over Q is a function s on some subset of Q such that $s(q) \subseteq U \mid U \in D_q$ wherever $s(q)$ is defined.
- A is a set of operators, where an operator $a \in A$ is a tuple $\langle\ name, parameters, precs, effects\ \rangle$. $parameters$ is a set of variables. $precs$ and $effects$ are partial assignments over $Q \mid \forall_{D_q \in Q} \forall_{\langle p, V \rangle \in D_q} V \subseteq parameters$. If the size of all the domains of the $parameters$ is exactly one we call the action *grounded*.
- s_0 is a state over Q called the initial state.
- s_g is a partial state over Q which satisfies the goal.

This is different from the encoding utilised by Fast Downward in which a value of a state variable is a single grounded atom (see Section 2.6.2). Therefore we changed the definition of a *partial variable assignment* over a state variable to mean any atom in any of the sets of the values.

Example 4.1.1 *TIM generates the following state variable for the type* block *in the* Blocksworld *domain (also depicted in Figure 4.9). The state variable contains the following values:*

- (clear block), (ontable block).
- (holding block).

- (on block block'), (clear block).
- (on block' block), (on block block").
- (ontable block), (on block' block).

Compare this to the state variables generated by Fast Downward (also depicted in Figure 4.8):

- (ontable block).
- (holding block).
- (on block block').

and

- (clear block).
- $\bot$.

As discussed in Section 2.4.2, encoding a planning problem using state variables also exposes the relationships between the values of a state variable. The relations between different values of a state variable and the transitions between them are encoded in a *lifted transition graph*. The dependencies between different state variables are encoded in a *causal graph*. In this section we present data structures that are closely related to the *domain transition graphs* (see Definition 17) and *causal graphs* (see Definition 18). However, in order to construct these structures we do not need to ground the entire domain. This makes our approach feasible for larger problem instances that cannot be encoded and solved using previous methods.

4.1.1 Constructing the lifted transition graph

We use TIM to extract state variables from a typed planning problem. Recall from Section 2.6.1 that property spaces define bags of properties (*states*) that are exchanged using transition rules. Attribute spaces, on the other hand, either increase or decrease the number of properties. Just as in the case of the transformation algorithm used by Fast Downward those state variables that do not *exchange properties* will be encoded with domains which can either be true or false.

The value domains of the state variables found by TIM are more concise than those found by Fast Downward. This is because TIM encodes the values of a state variable as a set of properties while Fast Downward encodes the values of a sate variable as either a grounded atom, $\top$, or $\bot$. We compile the values of state variables and the transitions between them in *lifted transition graphs*. These graphs are constructed using the same rules as DTGs. However, where DTGs only contain a single grounded atom per node, a node in a lifted transition graph contains a set of lifted atoms.

Definition 47 — Lifted Transition Graph

A lifted transition graph is a labeled bidirectional graph that encodes the values of a state variable. The edges between these values encode how those values can be altered. Lifted transition graphs are created for property spaces and attribute spaces.

Given a property space $\langle P, T, S, C \rangle$, a node is created for every state $s \in S$. Each property $p_i \in s$ is transformed into an atom $a = \langle p, V \rangle$ and added to a node, where

- $p = \langle name, types \rangle$ is the predicate associated with p_i.
- For each variable $v_j = \langle t, D_v \rangle \in V$ the type t is equivalent to the corresponding type of p and the value of the domain D_v is equal to all the objects whose type is either the same as or a super type of t. Except for $j = i$; in that case $D_v = C$.

A transition between two nodes n_{from} and n_{to} is added if there exists a transition rule $enablers \overset{op}{\Rightarrow} start \rightarrow finish \in T$ such that $start$ is the bag of properties n_{from} is created from and $finish$ is the bag of properties n_{to} is created from. The transition is labeled $op_{precs} \setminus n_{from}$.

A lifted transition graph is constructed identically for an attribute space $\langle P, T, C \rangle$, except for the transformation of a property $p_i \in s$ is into an atom $a = \langle p, V \rangle$. The transformation is performed as follows:

- $p = \langle name, types \rangle$ is the predicate associated with p_i.
- For each variable $v_j = \langle t, D_v \rangle \in V$ the type t is equivalent to the corresponding type of p and the value of the domain D_v is equal to all the objects whose type is either the same as or a super type of t.

An extra empty node is added that is used to add transitions when then the $start$ or $finish$ bag is empty.

As stated before, this structure is reminiscent of a DTG, but because we allow nodes to contain multiple atoms it is a more concise and informative structure. We will expand on this when we explain how the lifted causal graph heuristic is calculated.

Example 4.1.2 *To show an example of how a* lifted transition graph *is constructed consider the following* property space *constructed by TIM for the* Driverlog *domain for the type* Driver.

- $P = at_1, driving_1$
- $T = \{\} \overset{walk}{\Rightarrow} \{at_1\} \rightarrow \{at_1\}, \{\} \overset{board}{\Rightarrow} \{at_1\} \rightarrow \{driving_1\}, \{\} \overset{disembark}{\Rightarrow} \{driving_1\} \rightarrow \{at_1\}$
- $S = [at_1], [driving_1]$
- $C = \langle driver3 \rangle$

The two properties at_1 and $driving_1$ are transformed into the following atoms, respectively: (at { driver3 } { s1, p1-2, s2}) *and* (driving { driver3 } { truck1, truck2 }). *Note that the set of objects which match the type* Driver *might contain more objects than* driver3*, but we limit the domain which matches the index of the properties to C.*

Lastly we add the transitions. For this property space the following transition rules exists:

$$\begin{array}{lll} \{\} & \overset{board}{\Rightarrow} & at_1 \rightarrow driving_1 \\ \{\} & \overset{disembark}{\Rightarrow} & driving_1 \rightarrow at_1 \\ \{\} & \overset{walk}{\Rightarrow} & at_1 \rightarrow at_1 \end{array}$$

These are transformed into the following transitions:

$$(at\{driver3\}\{s1, p1-2, s2\}) \underset{board}{\overset{\{\}}{\rightarrow}} (driving\{driver3\}\{truck1, truck2\})$$
$$(driving\{driver3\}\{truck1, truck2\}) \underset{disembark}{\overset{\{\}}{\rightarrow}} (at\{driver3\}\{s1, p1-2, s2\})$$
$$(at\{driver3\}\{s1, p1-2, s2\}) \underset{walk}{\overset{\{\}}{\rightarrow}} (at\{driver3\}\{s1, p1-2, s2\})$$

We do not need to encode *all* the attribute spaces found by TIM. If an attribute space is subsumed by a property space we ignore it.

Definition 48 — Subsumed spaces
Given two property or attribute spaces s and s', we say that s subsumes s' *iff* we can find an injective and non-surjective function between the sets of properties s'_P and s_P such that the following properties hold:

- The name of the associated predicate of every property $p \in s_P$ is the same as the name of the associated predicate of the mapped property $p' \in s'_P$.
- Given the types T of the associated predicate of every property $p \in s_P$ and the types T' of the associated predicate of the mapped property $p' \in s'_P$ then every type T_i is equal to or a supertype of T'_i, where $i \in \{0, 1, \ldots, \mid T \mid\}$.

Example 4.1.3 *For the* driverlog *domain TIM finds the following property state for the type* driver*:*

- $P = at_1, driving_1$
- $T = \{\} \overset{walk}{\Rightarrow} \{at_1\} \rightarrow \{at_1\}, \{\} \overset{board}{\Rightarrow} \{at_1\} \rightarrow \{driving_1\}, \{\} \overset{disembark}{\Rightarrow} \{driving_1\} \rightarrow \{at_1\}$
- $S = [at_1], [driving_1]$
- $C = drivers$

This property state subsumes the following attribute space that is also constructed by TIM:

- $P = at_2$
- $T = \{at_2\} \overset{board}{\Rightarrow} \{at_2\} \rightarrow \{\}, \{at_2\} \overset{disembark}{\Rightarrow} \{\} \rightarrow \{at_2\}, \{path_1\} \overset{walk}{\Rightarrow} \{at_2\} \rightarrow \{\}, \{path_2\} \overset{walk}{\Rightarrow} \{\} \rightarrow \{at_2\}$
- $C = drivers$

Following Definition 48 the following sets of predicates are associated with the property and attribute space, respectively: $\{\langle at, \{driver, location\}\rangle, \langle driving, \{driver, truck\}\rangle\}$ *and* $\{\langle at, \{driver, location\}\rangle\}$. *We can make an injective and non-surjective mapping from the latter predicate the the first predicate of the former set. Since both names and types are the same we conclude that the property space subsumes the attribute space and the latter can therefore be ignored.*

When we compare our encoding of a *Blocksworld* domain (Figure 4.9) with the encoding of Fast Downward (Figure 4.8) we observe that our encoding is more concise and consists of fewer graphs. Our encoding remains the same even if we add more blocks to the domain, whereas the Fast Downward encoding would create a new DTG for each block added. This means that our encoding can work on much larger problem instances, because our data structure does not increase as more objects are added, whereas Fast Downward creates $2n+2n^2$ nodes where n is the number of *Block* objects in the domain.

4.1.2 Causal Graph

Where lifted transition graphs denote the transitions between values of state variables we reuse a structure used by Fast Downward to denote the interactions and dependencies between state variables. We construct the exact same structure, the *causal graph*.

Definition 49 — Causal Graph

Given a typed planning task $\Pi = \langle T, O, P, A, s_0, s_g\rangle$ and a set of state variables, a causal graph is a bidirectional graph where the set of state variables is the vertex set. An edge (v, v') exists *iff* $v \neq v'$ and one of the following conditions holds:

- The lifted transition graph of v has a transition with some precondition on v'.
- The set of affected variables in the effect list of some action $a \in A$ affects both v and v'.

This data structure is exactly the same as its Fast Forward equivalent. However, as demonstrated by Example 4.1.1, our method creates fewer graphs to represent all the state variables. Consequently the causal graph will contain fewer nodes and fewer potential cycles that need to be broken.

4.1.3 Merging Lifted Transition Graphs

We can reduce the number of graphs even further by merging lifted transition graphs. This has two benefits. First of all, by merging two lifted transition graphs we decrease the number of graphs by one. Secondly, by merging lifted transition graphs we could potentially reduce the number of cycles in the causal graph without having to remove any transitions or ignore any preconditions. However, we must be careful because merging lifted transition graphs expands the size of the graph exponentially, as shown by Merge and Shrink (see Section 2.4.2). This is less of a concern to our approach because the lifted transition graphs are constructed based on *types* not *objects*.

Proposition 4.1.1 *We merge any pair of lifted transition graphs whose property or attribute spaces apply to the same set of objects.*

Given two lifted transition graphs $\langle V, E \rangle$ *and* $\langle V', E' \rangle$ *we construct a new lifted transition graph where the nodes are created by taking the Cartesian product of* V *and* V'*:* $V'' = \{v \cup v' \mid v \in V \wedge v' \in V'\}$*. For every pair of nodes* $n_o \in V''$ *– which have been created by merging* $v_0 \in V$ *and* $v'0 \in V'$ *– and* $n_1 \in V''$ *– which have been created by merging* $v_1 \in V$ *and* $v'_1 \in V$ *– we copy any transition* $e \in E \cup E'$ *that has either* v_0 *or* v_1 *as the start node and* v'_0 *and* v'_1 *as the end node* iff *the following properties hold:*

- *There are no preconditions* $p \in e_{precs}$ *which are mutex with any atom in* n_0*.*
- *There are no effects* $f \in e_{effects}$ *which are mutex with any atom in* n_1*.*
- *For any atom in* $f \in n_0$ *that is not affected by the transition there must be an atom* $f' \in n_1$ *that is identical to* f*.*
- *For any atom in* $f \in n_1$ *that is not added by the transition there must be an atom* $f' \in n_0$ *that is identical to* f *that is not removed by the transition.*

The label of the new transition e *is* $e_{precs} \setminus v_0$*.*

In the last two cases of Proposition 4.1.1, f and f' are called *persistent* because they are not affected by the transition.

Example 4.1.4 *Consider a Driverlog problem; the constructed* lifted transition graphs *are depicted in Figure 4.1. Two lifted transition graphs are constructed based on property spaces for the type truck. One encodes the location of the truck with a single transition* Drive *and the other encodes if the truck is empty or is being driven with two transitions* Debark *and* Board*.*

By merging *these lifted transition graphs we end up with the following nodes:*

- { (empty { truck1, truck2 }), (driving {driving1} { truck1, truck2 }) }.
- { (at { truck1, truck2 } {s0, s1, p0-1}), (driving {driving1} { truck1, truck2 }) }.

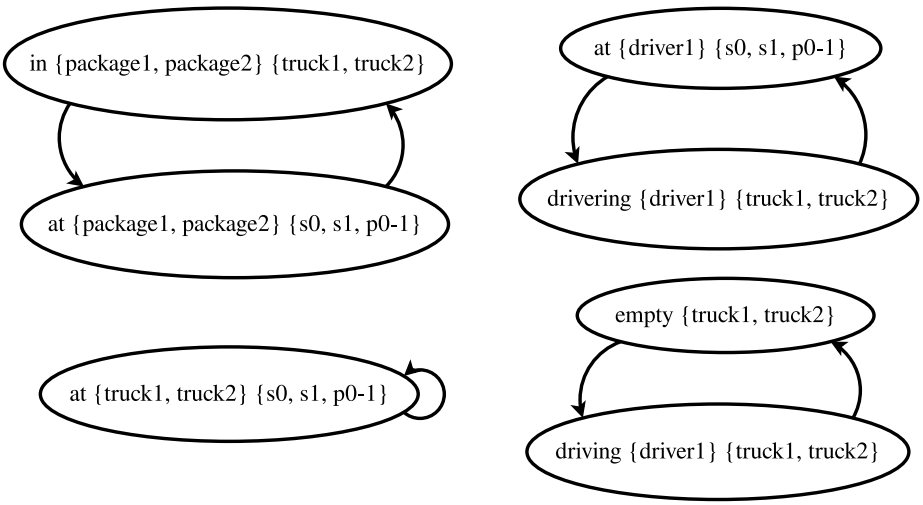

Figure 4.1: Lifted transition graph for a Driverlog domain.

Next we need to copy all the transitions but make sure that the preconditions and effects do not violate any constraints from Proposition 4.1.1. So for example, consider the latter node

$$\{(at\{truck1, truck2\}\{s0, s1, p0 - 1\}), (driving\{driving1\}\{truck1, truck2\})\}$$

and the transition Drive. *We cannot connect this transition with*

$$\{(empty\{truck1, truck2\}), (driving\{driving1\}\{truck1, truck2\})\}$$

as the end point, because the fact (empty { truck1, truck2 }) *is not achieved by this transition.*

By merging the two lifted transition graphs we create a new state variable which records both the location of the truck and whether or not it is being driven. The merged lifted transition graph is depicted in Figure 4.2. While it increases the number of nodes (possibly exponentially: see Section 2.4.2), it removes the cycle in the causal graph between the unmerged state variables. This means that less information will be lost and this will translate into more informative heuristic estimates. We need to be careful, since we could run into the problem that we do not have enough memory to hold the merged structures. If this is the case then we do not merge the state variables.

4.1.4 Grounding the Lifted Transition Graphs

Example 4.1.4 shows that we create fewer lifted transition graphs compared to the number of DTGs constructed by *Fast Downward* and the causal graph constructed contains

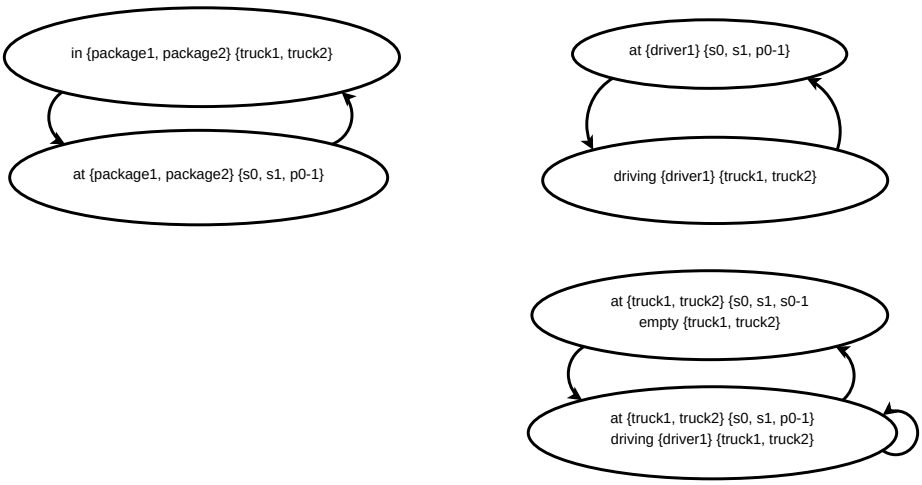

Figure 4.2: Merged lifted transition graph for a Driverlog domain.

fewer nodes and cycles. However, because the transitions in the lifted transition graphs are constructed based on transition rules no problem specific constraints are taken into account. For example, in Figure 4.2 the lifted transition graphs show that a driver can walk to different locations and board a truck, a truck can drive to different locations, etc. However, it is not shown which locations are actually connected. If none of the locations are connected then the drive and walk actions are never applicable so no driver or truck can reach any location other than the one it started at. But this information is not apparent in a lifted transition graph. In order to expose this information we ground a limited set of domain variables.

This is also necessary in order to be able to split the original problem into a set of $SAS^+ - 1$ sub-problems and being able to solve them. The *lifted causal graph* heuristic adapts the incomplete algorithm used by Fast Downward to extract a heuristic estimate. This algorithm is incomplete because it greedily tries to find plans to reach the *higher level values*. Once a higher level value has been reached the algorithm will not try to find a different solution to reach that value but commits to the assignments made to the the *lower level variables*. If we consider the lifted transition graphs in Figure 4.2 it is clear that an adaptation of this algorithm will not be very effective.

For example, if we want to reach the goal *(at package1 s1)* and the value of the variable domain of *package1* in the initial state is *(at package1 s0)* then this problem is unsolvable because the goal and the initial state share the same node.

To solve this problem we want to split objects into sets such that all the objects in a set can become *equivalent*, as we have done in Section 3. This way, using the lifted transition graphs from Figure 4.2 as an example, we would split all the locations (*s0*, *s1*, and *p0-1*) into separate sets. The result is shown in Figure 4.3. When we try to solve the $SAS^+ - 1$ sub-problem then – starting from the node *(at { package1, package2 } s0)* – we need to find a way to reach the value *(at { truck1, truck2 } s0)*. We do not care about the identity of the truck that reaches the location *s0*. Instead of considering

all the possible transitions (*(load package1 truck1 s0)* and *(load package1 truck2 s0)*) individually we initialise the lifted transition graph for the type Truck with all possible initial values and return the shortest plan which reaches the fact *(at { truck1, truck2 } s0)*.

Theorem 4.1.1 *If two objects o and o' are equivalent then Theorem 3.2.1 tells that if a fact f is reachable then $\kappa_f(o, o')$ is reachable too – under the delete relaxation. This theorem extends to $SAS^+ - 1$ sub-problems. Given a node n for a set of atoms N that can be initialised with set of grounded atoms G, then any node reachable from N can also be reached if n is initialised with the set $\{\kappa_g(o, o') \mid g \in G\}$. The reason is that all the preconditions of any transition of the* high-level variable *are not dependent on o or o'.*

Theorem 4.1.1 allows us to create a single lifted transition graph for a set of *equivalent* objects instead of creating a different lifted transition graph per object. If the objects can become equivalent then all the objects of the lower level variables of a $SAS^+ - 1$ problem can reach any value because all the preconditions on external state variables are removed. So if we do not care about the *identity* of the object that reaches a certain value, we do not have to use separate graphs to find a solution. If we do care about the identity then we must ignore all the other objects who do not match the identity of the object we are interested in.

We can create these sets as follows: given a planning problem Π and a state s, create a lifted RPG until the level-off point and split the objects as they appear in the equivalent object sets at the level-off point. However, we want to avoid the overhead of generating a lifted RPG per state. So instead we approximate the sets of objects which can be equivalent as follows:

Definition 50 — Potential equivalence

Given a planning problem Π and two objects $o \in \Pi_O$ and $o' \in \Pi_O$ we say that o and o' can potentially be made equivalent if the following properties hold:

- For any *static* fact $a \in \Pi_{s_0}$ there is a fact $\kappa_a(o, o')$ that is part of Π_{s_0}.
- Both objects are part of same property spaces.
- Both objects are part of at least one property space.

This definition overestimates the set of objects that could potentially become equivalent. For example, consider a *Driverlog* domain where the road network is such that two trucks at different locations cannot reach each other. If we build a lifted RPG then the trucks are split up accordingly, but our estimation groups them together. To circumvent this problem, we post process the generated equivalent object sets by checking if a lifted transition graph is split up into separate graphs. If this is the case then we split objects if they are not part of the same graph.

This does not circumvent all the problems. For example, given a graph it is still possible that a path exists from a node n to another node n' but there is no path from n' to n. A truck located at n can reach the initial fact of the truck at n' but the reverse is not true. So these objects can never become equivalent, but we still mark them as potentially equivalent.

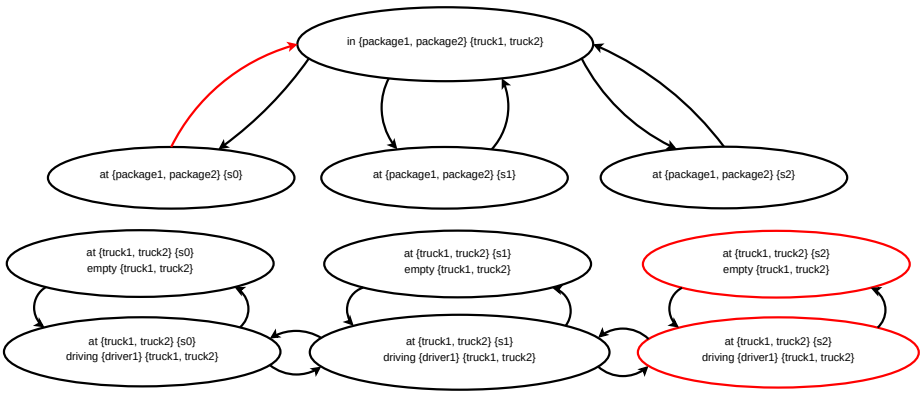

Figure 4.3: $SAS^+ - 1$ problem for a driverlog problem.

4.1.5 Splitting up Lifted Transition Graphs

In order to calculate the heuristic estimate we reuse the $SAS^+ - 1$ structure as defined in Section 2.4.2.

Definition 51 — $SAS^+ - 1$

A $SAS^+ - 1$ task is a typed multi-valued planning problem Π with a designated variable $q \in \Pi_Q$ such that $CG(\Pi)$ has an edge from q to all other variables $q' \in \Pi_Q$, and no other arcs. This variable q is called the *high-level* variable, whereas all other state variables are called *low-level* variables. A goal must be defined for the high-level variable, and goals must not be defined for the low-level variables.

The definition is the same as in Section 2.4.2, except that the values of the state variables are sets of atoms instead of a single grounded atom. This means that to solve a *high-level* variable we sometimes have a disjunction of goals defined for a *low-level* variable. This is a consequence of how the lifted transition graphs are constructed. For example, consider the $SAS^+ - 1$ problem depicted in Figure 4.3. The goal for the *high-level* variable is *(at package s2)* and the initial value is *(at package s0)*. The precondition to get a package in a truck from the initial state is *(at truck s0)*. The *low-level* variable of truck contains two values which satisfy that precondition: $\{$ *(at* $\{$*truck1, truck2*$\}$ $\{$*s2*$\}$*)*, *(empty* $\{$*truck1, truck2*$\}$*)* $\}$ and $\{$ *(at* $\{$*truck1, truck2*$\}$ $\{$*s2*$\}$*)*, *(driving* $\{$*driver1*$\}$ $\{$*truck1, truck2*$\}$*)* $\}$.

As discussed in Section 2.4.2, solving the $SAS^+ - 1$ task is an NP-complete problem. This complexity does not change for our encoding, because – despite having different values for the state variables – the problem remains the same. To solve the $SAS^+ - 1$ task Fast Downward uses an incomplete algorithm; we use the same approach and use a similar algorithm. The algorithm used by Fast Downward does not reevaluate a node once it is reached and commits to whatever actions were executed to reach it. As we have discussed in Section 2.4.2 this can lead to situations where we cannot find a solution for the $SAS^+ - 1$ task even if the task is solvable.

In order to use a similar incomplete algorithm and not run into many dead ends

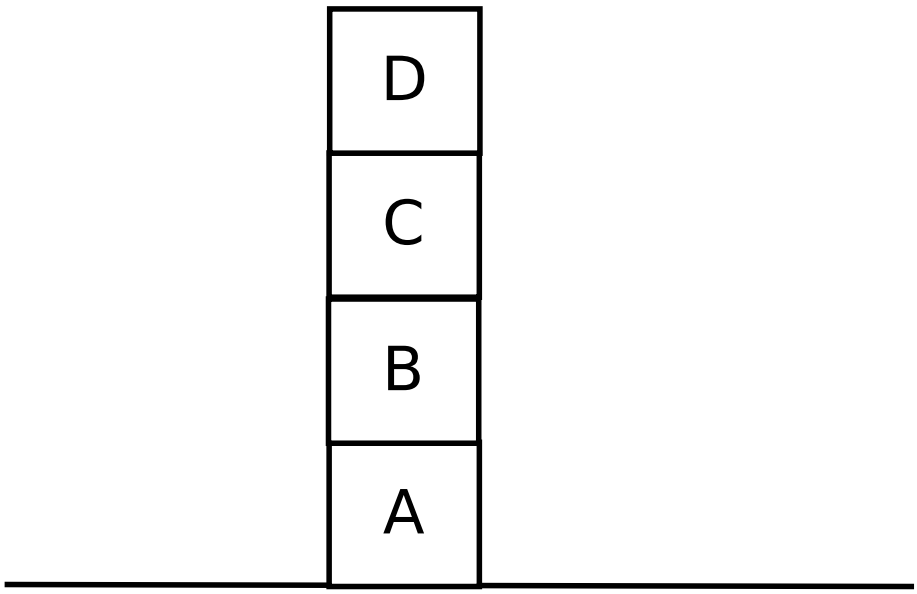

Figure 4.4: Initial state for a blocksworld problem.

we need to update the lifted transition graphs. The reason is that the lifted transition graphs constructed using the procedure discussed will not be able to solve certain kinds of problems. For example, consider the lifted transition graphs for the driverlog domain depicted in Figure 4.2. If the value of the state variable in the initial state is *(driving driver1 truck1)* and the goal value is *(driving driver1 truck2)*, then – assuming a node cannot be reevaluated – this goal cannot be satisfied. This problem becomes even more pronounced when we consider the lifted transition graph for the *Blocksworld* domain as depicted in Figure 4.9. For example, consider the initial state depicted in Figure 4.4. The blocks B and C are mapped to the value $\{$ *(on block block), (on block block)* $\}$; if the goal is to get another block on top of any of these blocks (e.g. *(on D B)*) then this goal is unattainable.

To overcome this problem, we extend the lifted transition graph by making duplicates of nodes in the lifted transition graph. We use the following process to split the nodes up.

Proposition 4.1.2 *Given a lifted transition graph $\langle V, E\rangle$ then for every node $v \in V$ that has not already been copied, we create a copy v_{copy}* iff *v contains an atom $\langle p, V\rangle$ that corresponds to property p_i and one of the variable domains $V_j \in V \mid j \in \{0, \ldots, \mid V \mid -1\} \wedge j \neq i$ contains more than one object.*

Next every transition $e \in E$ that has v as the start or end node is copied where $v \in e$ is substituted by v_{copy}. Unless the other node $v' \in e \mid v' \neq v$ shares an atom $\langle p, V\rangle$ that corresponds to property p_i that is (1) not affected by e and (2) one of the variable domains $V_j \in V \mid j \in \{0, \ldots, \mid V \mid -1\} \wedge j \neq i$ contains more than one object. If that is the case then we create a copy v'_{copy} and copy e where v is substituted

by v_{copy} and v' is substituted by v'_{copy}. This method is called recursively, i.e. we search for any transitions which have v' as a start or end node and create the same copies.

Example 4.1.5 *Consider the lifted transition graph of the Blocksworld problem depicted in Figure 4.9. When we apply the above algorithm we end up with the graph depicted in Figure 4.5. The node*

$$\{(on\{b1, b2, b3\}''\{b1, b2, b3\})(on\{b1, b2, b3\}\{b1, b2, b3\}')\}$$

shares the fact (on { b1, b2, b3 } { b1, b2, b3 }') *with the node*

$$\{(clear\{b1, b2, b3\}\{b1, b2, b3\}')(on\{b1, b2, b3\}\{b1, b2, b3\}')\}.$$

Therefore we create a copy of the former and copy the transitions.
An more interesting example is the node

$$\{(clear\{b1, b2, b3\})(on\{b1, b2, b3\}\{b1, b2, b3\}')\}.$$

This node needs to be copied because of the variable domain { b1, b2, b3 }' . *The transitions between the node*

$$\{\}(holding\{b1, b2, b3\})\}$$

are copied, but we must also make copies of the node

$$\{(on\{b1, b2, b3\}''\{b1, b2, b3\})(on\{b1, b2, b3\}\{b1, b2, b3\}')\},$$

because the fact (on { b1, b2, b3 } { b1, b2, b3 }') *is shared and contains the variable domains* { b1, b2, b3 }' .

Returning to the previous example with the initial state depicted in Figure 4.4, to get another block on block B we can execute the sequence of actions (unstack A B), (stack x B). *If we want B to be stacked on another block and get another block on B than we can execute the sequence of actions* (unstack A B), (unstack B C), (stack B x), (stack y B)*, where $x \in \{A, B, D\}$ and $y \in \{B, C, D\}$. For all these sequences of actions we do not reevaluate any node as we did before.*

We denote the set of all the copies that have been made for a node n as $copies(n)$.

This method does increase the number of nodes in a lifted transition graph; in the worst case it can extend the number of nodes exponentially. For example consider the lifted transition graph in Figure 4.6. In this case we have a state variable that can take a large number of values and each transition between values either adds or removes an atom from the set of values but otherwise stays the same. The split up lifted transition graph is depicted in Figure 4.7 and contains $!n$ nodes, where n is the number of nodes in the original graph.

The scenario depicted in Figure 4.6 is not present in any of the benchmark domains and any reasonable planning problem is unlikely to exhibit this scenario. If such a scenario does show up we can forego splitting up the lifted transition graph, but this will affect the heuristic estimate, as we shall discuss in the next section.

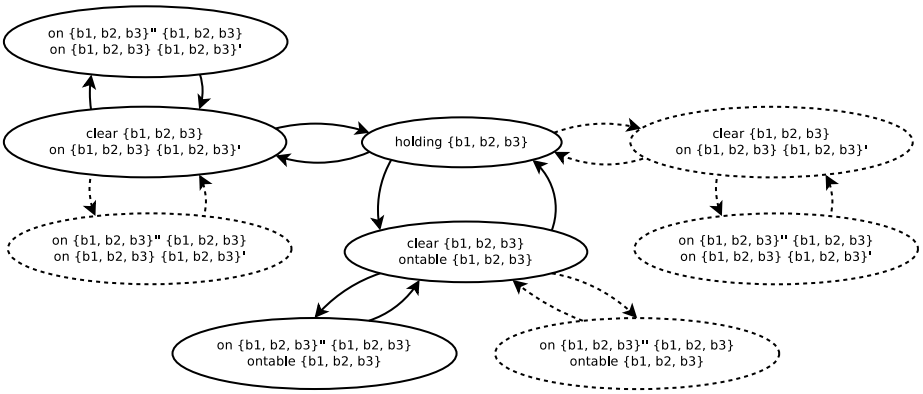

Figure 4.5: Split up lifted transition graph for Blocksworld; the dotted nodes and transitions are copies.

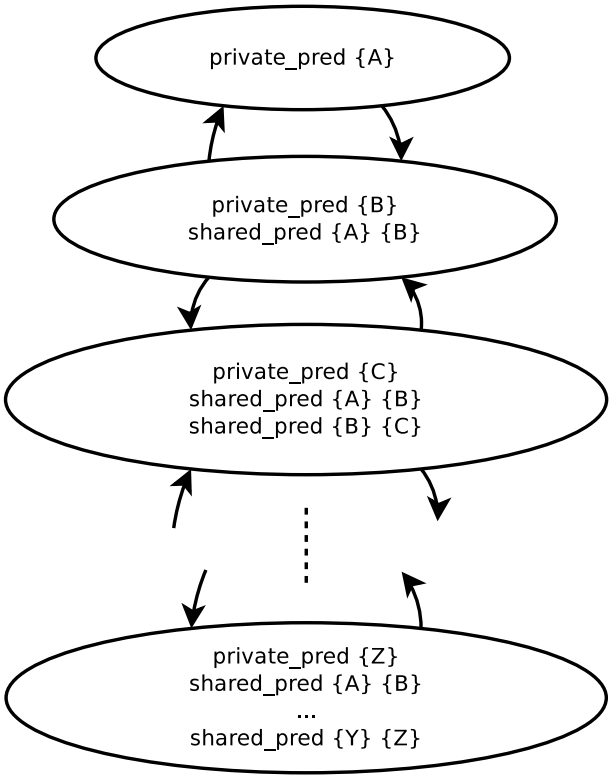

Figure 4.6: Lifted transition graph that splits up into an exponential number of nodes.

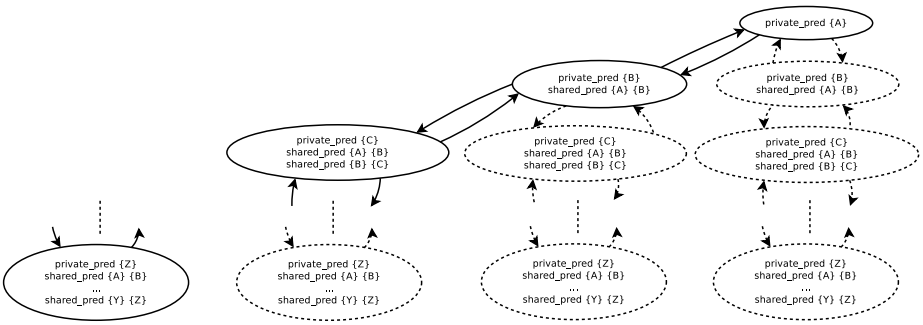

Figure 4.7: Lifted transition graph splitt up into an exponential number of nodes.

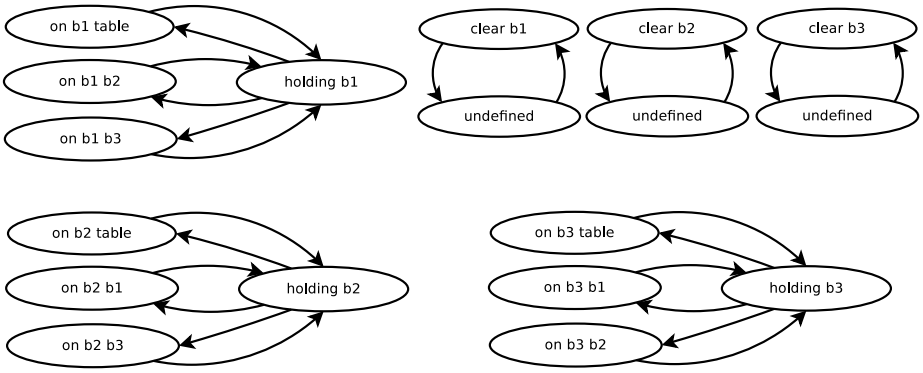

Figure 4.8: Domain transition graphs for a Blocksworld domain.

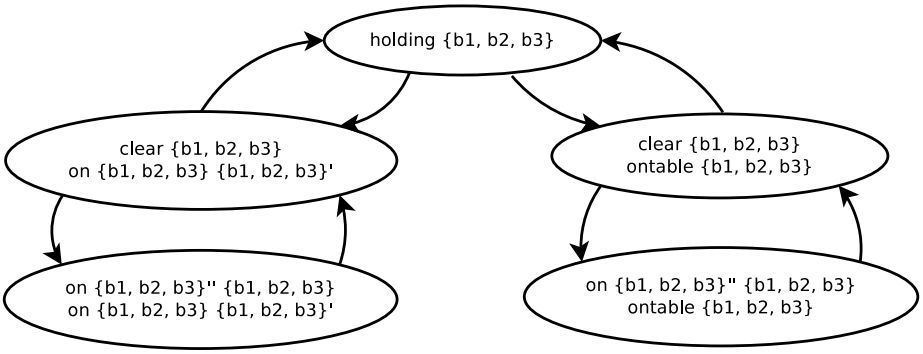

Figure 4.9: Lifted transition graphs for a Blocksworld domain.

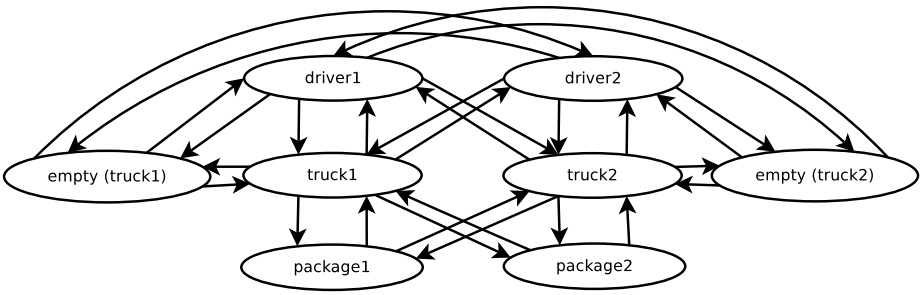

Figure 4.10: Causal graph as constructed by Fast Downward for the Driverlog domain.

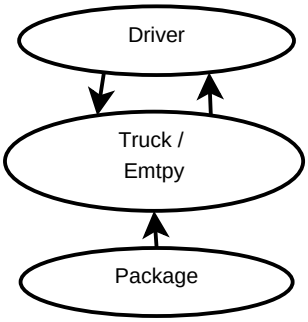

Figure 4.11: Causal graph as constructed by our method.

4.1.6 Breaking cycles in the causal graph

In order to reduce the original planning problem into a set of lifted $SAS^+ - 1$ tasks we need to break cycles in the causal graph. Because the data structure is the same as the one utilised by Fast Downward we use the same method as described in Section 2.4.2 to break these cycles. The causal graph for lifted transition graphs usually contains fewer nodes than its grounded equivalent. This is because we allow lifted transition graphs to be *merged* and because of the more concise encoding of TIM. For example, the causal graph for the lifted transition graphs of *Driverlog* (see Figure 4.2) contains only three nodes and does not contain the cycles between the state variables of a truck being empty. The causal graphs for the *Driverlog* domain are depicted in Figure 4.10 and Figure 4.11.

An important difference with the causal graph constructed by Fast Downward is that a DTG is constructed per *object*, whereas we construct a lifted transition graph per *type*. This means that dependencies between objects of the same type are pruned by our search algorithm. For example, in the *Blocksworld* domain we prune all dependencies between blocks and in the *Depots* domain we ignore dependencies between crates. This is the price we pay for using lifted structures.

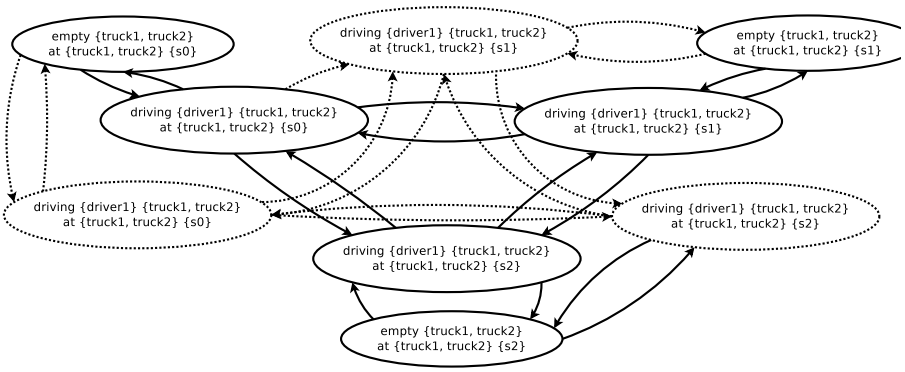

Figure 4.12: Split lifted transition graph for the type Truck for a Driverlog domain.

4.2 Calculating the heuristic

Now that we have explained how the lifted transition graphs and causal graphs are constructed we will now discuss how we use these structures to compute the lifted causal graph heuristic h^{lcg}. There are some important characteristics of the h^{lcg} heuristic that differ from the h^{cg} heuristic and affect the way we calculate it. First of all, the nodes of lifted transition graphs can contain more than one atom and the atoms that are part of a node do not need to be grounded. This means that given a goal, multiple values of a state variable could satisfy that goal.

Example 4.2.1 *Given the goal* (empty truck) *then the lifted transition graph in Figure 4.12 contains three nodes that satisfy that goal.*

To reduce the original problem into a set of $SAS^+ - 1$ tasks we need to break cycles in the causal graph. State variables are created for *types* instead of *objects*, which means that any dependencies between state variables of the same type are ignored. This also means that objects of the same *type* share the same lifted transition graphs if they are *potentially equivalent*. This means that we need to evaluate fewer graphs, which could allow us to calculate the heuristic estimate more quickly, because the number of lower-level variables in the $SAS^+ - 1$ task will be lower. On the other hand, we will have to do more work because we work with lifted atoms. We will discuss this trade-off when we present the results.

Definition 52 — Lifted $SAS^+ - 1$ task

Given a typed multi-valued planning problem $\Pi = \langle T, O, P, Q, A, s_0, s_g \rangle$, the causal graph $CG(\Pi)$, and a disjunctive set of atoms G where each atom $g \in G$ is part of the same state variable $q \in Q$, we define a lifted $SAS^+ - 1$ task as a typed multi-valued planning problem $\Pi' = \langle T, O, P, Q', A', s_0, G \rangle$ where $Q' = \{q' \in Q \mid q' = q \vee (q, q') \in CG(\Pi)\}$ and A' contains the same operators as A; however, every precondition or effect of any action $a' \in A'$ which makes an assignment to a state variable $q'' \notin Q'$ is removed.

If there are multiple variable domains containing a goal variable, then we select the state variable with the lowest number of dependencies given the acyclic causal graph CG. For example, given the goal *(driving {truck1, truck2} {driver1})* then there are two lifted transition graphs, one depicted in Figure 4.14 for the type *Driver* and the other depicted in Figure 4.12 for the type *Truck*. Depending on how the cycles in the causal graph are broken we prefer one over the other. If state variable for the type *Driver* is not dependent on the state variables of the type *Truck* then we pick the state variable of type *Driver* as the high-level variable. Otherwise we pick the state variable for the type *Truck*.

Informally we define a lifted $SAS^+ - 1$ task as a subset of the original planning problem where the goal is given as a disjunction of atoms. The set of goals G must be a subset of the associated finite domain of one of the state variables $q \in Q$ (i.e. $G \subseteq D_q$). The set of state variables considered is a subset $Q' \subseteq Q$, i.e. only the state variables which are either the *high-level variable* q or those on which q is directly dependent. The other state variables are ignored, the preconditions and effects of the operators are updated accordingly. Any precondition or effect that is not part of any of the associated finite domains $D'_q \mid q' \in Q'$ is removed.

4.2.1 Solving the lifted $SAS^+ - 1$ task

The differences with the original $SAS^+ - 1$ definition (see Definition 19) are that (1) the goal is no longer a grounded atom but rather an atom that does not have to be grounded and (2) the initial state is not a conjunction of grounded atoms, but rather a conjunction of atoms which do not have to be grounded. The original algorithm (see Algorithm 1) finds the cheapest path from the start node to the goal node. However, because the goal and initial atoms can be lifted in our case there might be multiple start and end points. Therefore we use an algorithm that finds a path from multiple source points to multiple destination points.

The values of state variables are a conjunction of atoms instead of a single grounded atom. To find the start nodes given an initial atom s we reuse the definition of *consistent* used to construct a lifted RPG (see Section 3.3).

Definition 53 — Consistent mapping of a lifted transition graph node

Given a state variable q and the lifted transition graph created for it $ltg(q)$, then we say that a set of atoms F is a *consistent mapping* to a node $n \in ltg(q)$ *iff* there exists a bijection between F and n such that for every atom $\langle p, V \rangle \in F$ and the atom it is mapped to $\langle p', V' \rangle \in n$ the following constraints hold:

- $p = p'$, i.e. the predicates must be the same.
- For every every pair of variables $\langle t, D_v \rangle \in V_i$ and $\langle t', D'_v \rangle \in V'_i$ the following constraints must hold:
  - $t = t'$, i.e. the types must be the same.
  - $D_v \subseteq D'_v$, i.e. the variable domain of the mapped atom must be a subset of the variable domain of the value of the state variable.

- If there is a *property* p_i associated with $\langle p', V' \rangle \in n$ then the variable domain D_i must be identical to any other variable domain for which a *property* is defined.

Example 4.2.2 *Given the node* { (at { truck1, truck2 } s1), (empty { truck1, truck2 }) } *then the following set of atoms:*

$$\{(at\ truck1\ s1), (empty\ truck2)\}$$

is not consistent, because the properties associated with the atoms – at_1 *and* $empth_1$ *respectively – imply that the first value of the atoms must be identical. So the following set of atoms is consistent:*

$$\{(at\ truck2\ s1), (empty\ truck2)\}$$

Given a lifted $SAS^+ - 1$ task $\Pi' = \langle T, O, P, Q', A', s_0, G \rangle$ we will present an algorithm that – like the algorithm used by Fast Downward to solve $SAS^+ - 1$ tasks – is also incomplete. In our algorithm we will refer to the *high-level variable* as $q_h \in Q'$ and the associated *lifted transition graph* as $ltg(q_h)$.

1. First of all we initialise our open queue $open$, the closed queue $closed = \emptyset$, and for every node $v \in ltg(q_h)$ that is not a copy we initialise the plan found to reach v. The plan found for v is denoted as $plan(v)$ and contains a sequence of actions. For every node v we search for a subset of atoms in the initial state $F \subseteq s_0$ that is *consistent* with the atoms in v. If such a subset if found, then we add the tuple $\langle v, F, s_0 \rangle$ to the queue $open$ and initialise $plan(v) = \langle\rangle$. In order to disable parts of the graph that will not be used, we add all the nodes $n \in ltg(q_h)$ that are copies to $closed$, except if $n \in copies(v)$.

2. Let $\langle v, F \rangle$ be an element of $open$ such that $v \notin closed$ and minimises the cost of $plan(v)$ and let s be the state that results from applying $plan(v)$ to s_0. First of all we add v to the closed list $closed$.

   We consider each transition t with action op that has v as the start node and v' as the end node. Every precondition $p \in op_{precs}$ that is part of v is initialised using the consistent set F. This step is necessary because *op* is a *lifted* action.

   Next we check if every other precondition $p \in t_{label}$ can be satisfied. For every precondition p that is part of the state variable q_l we construct a new lifted $SAS^+ - 1$ task where q_l is the *high-level variable* and the state variables are restricted to q_l. The initial state is equal to s and the goal is p. Note that we ignore all the dependencies of q_l so the solution can be found by finding the shortest path in the *lifted transition graph* from any of the source nodes for which a consistent set can be found from all the facts in s and one of the destinations nodes which contains a goal node $p' \in p$. After a plan is found which reaches p we update the variable domains of op and repeat this process until we have found a plan for each precondition.

   If there is a precondition $p \in t_{precs}$ for which no solution can be obtained we stop and continue with the next transition. Otherwise, let $plan(t_{precs})$ be the

plan found to solve all the preconditions t_{precs}, and let $\pi_{v'}$ be the concatenation of $plan(v)$, $plan(t_{precs})$, and $\langle op \rangle$. If $\pi_{v'} < plan(v')$ then $plan(v') = pi_{v'}$ (undefined plans have cost ∞). We define s as the state that results from applying the sequence of action $\pi_{v'}$ to the state s_0. Finally we add $\langle v', F', s \rangle$ to $open$.

3. Repeat the previous step until there are no more items in $open$ – in which case there is no solution – or if we remove $\langle v, F \rangle$ from the queue such that one of the goals $g \in s_g$ matches with an atom $f \in F$ – in which case we return $plan(v)$ as the solution.

This algorithm is more complex than the algorithm used by Fast Downward. The main reason is that the operators in the *lifted transition graphs* are lifted and need to be instantiated and updated every time a solution for a precondition is found. In addition, the atoms in the initial state and the goal atoms are not necessarily *grounded* so we might have to solve problems that have multiple source points and multiple end points.

Example 4.2.3 *Consider a* Driverlog *problem where the object* package1 *is located at location* s1 *and the goal is to get the package at* s2*. The domain contains two trucks;* truck1 *– which is empty and at location* s0 *– and* truck2 *– which empty and is at location* s2*. We denote the set of facts which are true in the initial state as* s_0*. For this example we ignore the drivers in this domain.*

Using the above algorithm we start by initialising the *open* queue and *closed* list for the *lifted transition graph* for the type package which is depicted in Figure 4.13. The *open* is initialised by the tuple $\langle \{$ *(at* $\{$ *package1, package2* $\}$ *s1)* $\}, \{$ *(at package1 s1)* $\}, s_0 \rangle$. The closed list contains the copy of the node $\{$ *(in* $\{$*package1, package2*$\}$ $\{$*truck1, truck2*$\}$*)* $\}$ because this node is not a copy of any of the nodes that are *consistent* with a subset of the initial state $F \in s_0$.

The only transition to consider (that does not have a node in the *closed* list) is *(load* $\{$*package1, package2*$\}$ $\{$*truck1, truck2*$\}$ *s1)*. This is a lifted transition, but we are not interested in loading *any* package in the truck but *package1*. Therefore we use the assignments made to the start node of the transitions to update the variable domains of the transition. In this case the precondition *(at* $\{$ *package1, package2* $\}$ *s1)* is updated to the value *(at package1 s1)*. Hence the transition becomes: *(load package1* $\{$*truck1, truck2*$\}$ *s1)*. Next we process the precondition *(at* $\{$ *truck1, truck2* $\}$ *s1)*. The *lifted transition graph* for the state variable this value belongs to is depicted in Figure 4.12.

We create a new lifted $SAS^+ - 1$ task where the goal is *(at* $\{$ *truck1, truck2* $\}$ *s1)*. Again we initialise the *open* queue and *closed* list. This time we add two items to the open queue: $\langle \{$ *(empty* $\{$ *truck1, truck2* $\}$ *),* $\{$ *(at* $\{$ *truck1, truck2* $\}$ *s0)* $\}, \{$ *(empty truck1)* , *(at truck1 s0)* $\} \rangle$ and $\langle \{$ *(empty* $\{$ *truck1, truck2* $\}$ *),* $\{$ *(at* $\{$ *truck1, truck2* $\}$ *s2)* $\}, \{$ *(empty truck1)* , *(at truck1 s2)* $\} \rangle$ and the closed list contains all the nodes that are copies.

The shortest path found is the sequence of actions $\langle$ *(board D truck1 s0)*, *(drive D truck1 s0 s1)* $\rangle$, where D could be any driver (they are ignored).

After we find this sequence of actions we find a solution to all the preconditions of the *load* transition. The goal found is *(at truck1 s1)*, therefore we update the variable domains of the transition which now becomes *(load package1 truck1 s1)* and add the

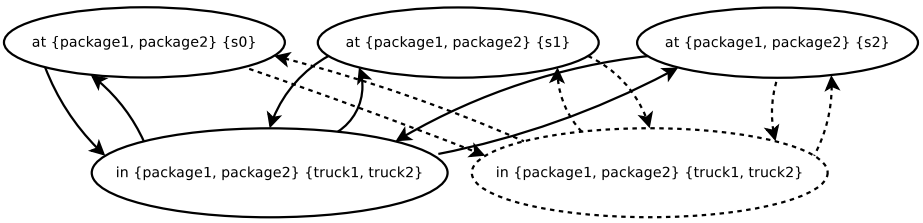

Figure 4.13: Split lifted transition graph for the type Package for a Driverlog domain.

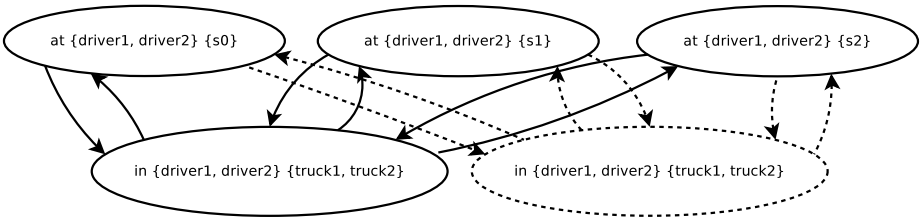

Figure 4.14: Split lifted transition graph for the type Driver for a Driverlog domain.

following tuple to the queue: $\langle\{$ *(in* $\{$ *package1, package2* $\}$ $\{$ *truck1, truck2* $\})\}, \{$ *(in package1 truck1)* $\}, s\rangle$, where s is the state that results from applying the sequence of actions: $\langle$ *(board D truck1 s0)*, *(drive D truck1 s0 s1)*, *(load package1 truck1 s1)* $\rangle$ to s_0. Note that this results in a state where all drivers are driving *truck1* simultaneously which is reflected in the fact *(driving D truck1)*. This is the next node we consider and we consider the transition *(unload* $\{$*package1, package2*$\}$ $\{$*truck1, truck2*$\}$ *s2)*. Again we update the variable domains of the transition which yields: *(unload package1 truck1 s2)*. The only precondition to resolve is *(at truck1 s2)*. Note that this time around we want a specific truck to reach the location *s2*. The $SAS^+ - 1$ task we construct will ignore the values of any other truck other than *truck1*.

So the *open* queue contains the tuple $\langle\{$ *(driving* $\{$ *driver1* $\}$ $\{$ *truck1, truck2* $\}$ *)*, *(at* $\{$ *truck1, truck2* $\}$ *s1)* $\}, \{$ *(driving driver1 truck1)* , *(at truck1 s1)* $\}\rangle$ and the closed set is empty. The solution to this problem is $\langle$ *(drive driver1 truck1 s1 s2)* $\rangle$. This satisfies all the preconditions of the *unload* transition and satisfies the goal.

So the final plan returned is $\langle$ *(board driver1 truck1 s0) (drive D truck1 s0 s1) (load package1 truck1 s1) (drive D truck1 s1 s2) (unload package1 truck1 s2)* $\rangle$ with cost 5.

In Section 4.1.5 we described an algorithm to make duplicates of lifted transition graph nodes. We do this to prevent our algorithm from concluding that no solution is possible, whereas h^{cg} can find a solution to the relaxed problem.

Example 4.2.4 *Consider a* Driverlog *problem where* driver1 *is driving* truck1 *and the goal is to get* driver1 *into* truck2*. The lifted transition graph for the type Driver is depicted in Figure 4.14. Note that if we exclude the copied nodes and transitions (i.e. those which are depicted with dotted lines) then there is no sequence of transitions – which do not reevaluate any nodes – to get the driver into another truck.*

4.2.2 Calculating the lifted causal graph heuristic

Now that we have discussed the algorithm used to solve a lifted $SAS^+ - 1$ task we will move on to describe the complete algorithm to calculate the lifted causal graph heuristic. Like Fast Downward we split the entire planning problem into a sequence of lifted $SAS^+ - 1$ subtasks where the only interactions are between a state variable and all the state variables it depends on. These dependencies are resolved by constructing an acyclic causal graph. The algorithm we use is a little more complex than the algorithm presented by Fast Downward, because the value domains of our state variables are a conjunction of atoms instead of a single grounded atom. This is a product of the translation algorithm we have used (TIM) and because we allow some lifted transition graphs to be *merged*.

We introduce a cost function $cost_q(D, d')$, where D is a value of the state variable q and d' is an atom that does not need to be grounded. Given a typed multi-valued planning task Π and an acyclic causal graph $CG(\Pi)$ then $cost_q(D, d')$ is calculated as follows.

We create a lifted $SAS^+ - 1$ task Π' where the values of the state variables are identical to Π_{s_0} except for (1) the value of the state variable D and (2) the goal is replaced by d'. Let π be the plan returned by solving Π'. $cost_q(D, d')$ is then the sum of the costs of the transitions in π, where the cost of the transitions of the *high-level variable* have cost 1 and the cost of a transition on a *lower level variable* q_l from $V \in q_l$ to v' has the cost $cost_{q_l}(V, v')$.

The lifted causal graph heuristic is then defined as the sum of $cost_q(D, d')$, where $d' \subseteq s_g$, $q \in Q$ where q is the state variable that contains a value $D_q \in q$ that contains the value d', and $D \subseteq s_0$ is the set of values from the initial state which are *consistent* with any of the values $v \in q$.

Example 4.2.5 *Consider a planning problem where the following values are true in the initial state.*

The planning problem contains four locations, $s1$, $s2$, $s3$, and $p2-3$. The locations $s2$ and $s3$ are fully connected, but $s1$ can only be reached from $s2$ and is a dead end. The location $p2-3$ is fully connected with the locations $s2$ and $s3$ with a path so only drivers can traverse these paths.

There are two drivers $d1$ and $d2$; the former is at location $s2$ and the latter is at location $p2-3$. Furthermore, there are two trucks $t1$ and $t2$. Both trucks are empty and $t1$ is at location $s1$ and $t2$ is at location $s3$. There is a single package $p1$ which is loaded in truck $t2$ and the single goal for this planning problem is to get $p1$ to location $s1$.

The lifted transition graph for the package $p1$ is depicted in Figure 4.15, the lifted transition graph for the trucks $t1$ and 2 is depicted in Figure 4.16, and the lifted transition graph for the drivers $d1$ and $d2$ is depicted in Figure 4.17. To make the graphs more easy to understand we have decided to not depict the copies of nodes which are not necessary. When we explain the example we will include these copies in the graphs when they are needed. The causal graph is made acyclic such that the packages are dependent on trucks and trucks are dependent on drivers.

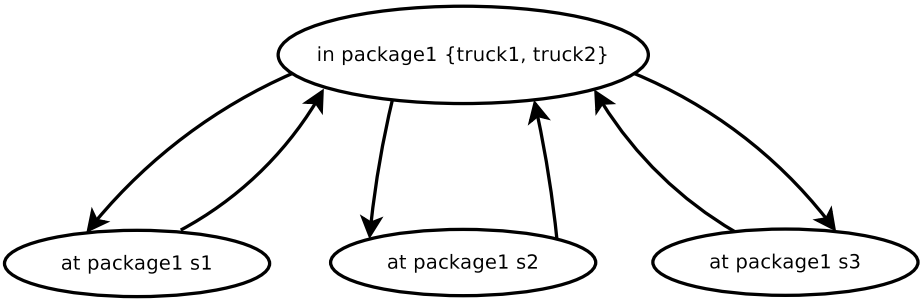

Figure 4.15: Split lifted transition graph for the type Package for a Driverlog domain for Example 4.2.5.

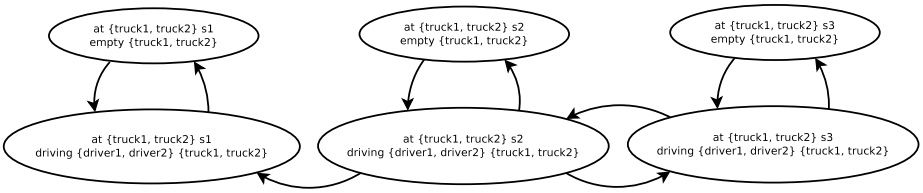

Figure 4.16: Split lifted transition graph for the type Truck for a Driverlog domain for Example 4.2.5.

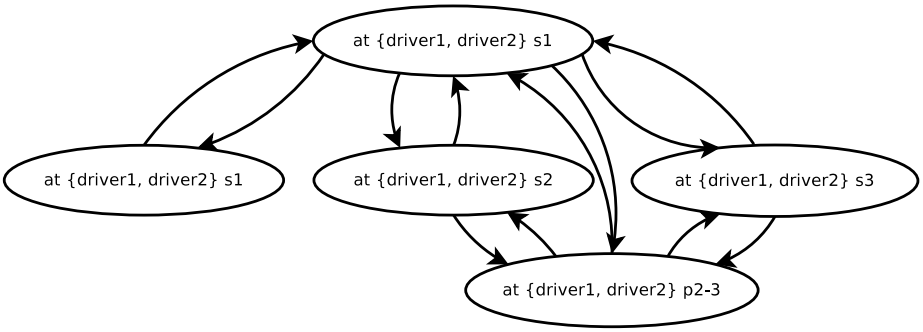

Figure 4.17: Split lifted transition graph for the type Driver for a Driverlog domain for Example 4.2.5.

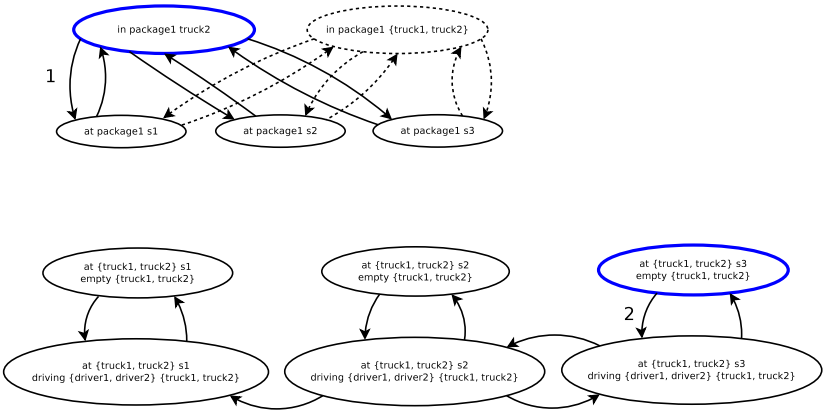

Figure 4.18: Lifted SAS^+ problem to get the package $p1$ inside the truck $t1$. The blue nodes are the starting values.

The goal we want to achieve is *(in package1 truck1)*; the lifted transition graph which can achieve this goal is depicted in Figure 4.15. The value which is true in the initial state for this state variable is *(in package truck2)*. We create a lifted SAS^+ problem which is depicted in Figure 4.18. This problem starts in the node coloured blue. This node corresponds with the value that is true in the initial state for $p1$ and this also means that a copy of this node is not added to the closed list. The goal we want to achieve is to get the package into the other truck, which means that we need to get the package to location $s1$ first. Other actions that get the package to any of the other locations will be evaluated by our algorithm but since they are irrelevant to achieve the goal we will ignore them.

To execute the action *(unload p1 t2 s1)* (marked with the numeral *1*) we need to satisfy the precondition *(at t2 s1)*. In this instance we want a specific truck to reach the location $s1$ because the package is already in this truck. The created SAS^+ problem executes the action *(board* $\{$*d1, d2*$\}$ *t2 s3)*. In order to calculate the cost of achieving this action we calculate $cost_q(v, v')$, where q is the state variable for the type *Truck*, v is the value that is true in the initial state s_0 for $t2$ and v' is $\{$ *(at t2 s3)*, *(driving* $\{$*d1, d2*$\}$ *t2)* $\}$. This problem is translated in a lifted SAS^+ task that is depicted in Figure 4.19. All the copies are added to the closed list so these are not depicted.

The sub-problem for getting any driver to the location $s3$ is a multi-source single destination problem. Because there are no dependencies for the drivers, the driver who is closest to the goal value will be chosen to board the truck. In this case the driver $d2$ is closest so the following sequence of actions is executed: $\langle$ *(walk d2 p2-3 s3)*, *(board d2 t2 s3)* $\rangle$. So the cost of board transition is 2. The value reached after boarding the driver is $\{$ *(at t2 s3) (driving d2 t2)* $\}$. After the driver has boarded the truck there are no longer any dependencies on the driver, because the fact that a driver is driving is encoded in the values of the variable domain of the type *Truck*. So the cost of achieving the goal *(at t2 s1)* is the cost of the sequence of actions $\langle$ *(board d2 t2 s3)*, *(drive t2 d2 s3 s2)*, *(drive t2 d2 s2 s1)* $\rangle$ which is equal to $2 + 1 + 1 = 4$.

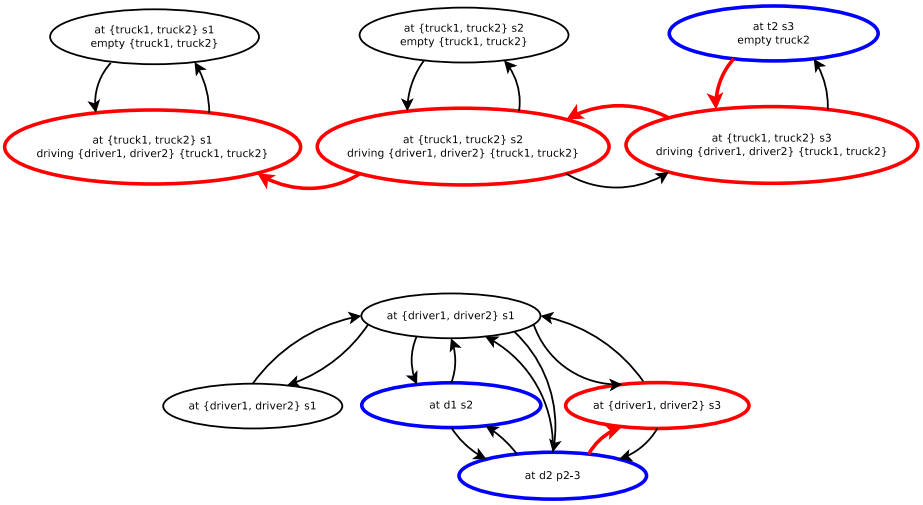

Figure 4.19: SAS^+ problem to get the truck $t2$ to the location $s1$. The blue nodes are the starting values and the red edges and nodes are the solution found.

Going back to the original SAS^+ problem depicted in Figure 4.19, all the preconditions of the unload transition are satisfied and the cost of this transition is equal to $4 + 1 = 5$. After reaching this node we update the state s_0 by applying the found sequence of actions: $\langle$ *(board d2 t2 s3)*, *(drive t2 d2 s3 s2)*, *(drive t2 d2 s2 s1)*, *(unload p1 t2 s1)* $\rangle$. Finally we try to get the package inside $t1$ by applying the action *(load p1 t1 s1)* whose preconditions are already satisfied, so the returned heuristic estimate is 6, which is the optimal heuristic.

4.3 Implementation of the planning system

The implementation of the planning system is identical to Algorithm 5 with the exception that we replace the heuristic h^{lrpg} with h^{cg}. The complete algorithm is listed in Algorithm 6.

4.4 Results

We have chosen the same setup as with the experiments carried out to test h^{lrpg}. We ran all our experiments on an Intel Core i7-2600 running at 2.4GHz and allowed 2GB of RAM and 30 minutes of computation time. We took seven problem domains from various planning competitions, and we now discuss the results obtained in these domains.

We configured the Fast Downward planner to use an eager, greedy heuristic in such a way that we only use the distance to the goal as the heuristic value and we do not take the distance from the start into consideration.

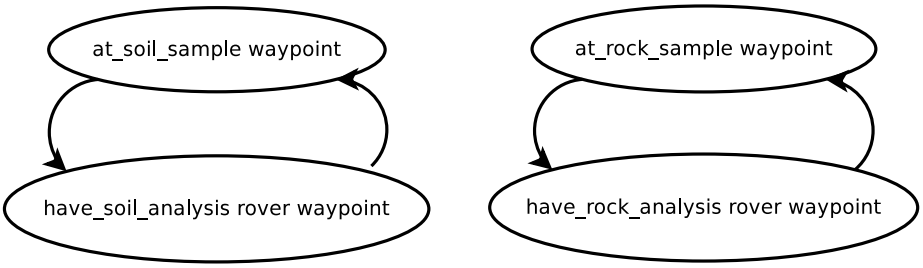

Figure 4.20: Unmerged lifted transition graphs for the soil and rock analysis property states.

4.4.1 Merging v.s. not merging

We want to test our intuition that merging lifted transition graphs improves the heuristic estimates due to preserving more information. Therefore we ran two instances of our planning system; one merges the lifted transition graphs while the other only uses the lifted transition graphs we create based solely on the property and attribute spaces constructed by TIM. The results are depicted in Table 4.1. As we can see, for most domains where we can merge lifted transition graphs merging provides better heuristic estimates and is able to solve five more problems than the encoding without merging.

The only odd result in this table is for the *Rovers* domain. The reason for this result is that merging property states creates an inefficient encoding of the property states: { *(at_rock_sample waypoint) (have_rock_analysis rover waypoint)* } and { *(at_soil_sample waypoint)*
(have_soil_analysis rover waypoint) }. If we do not merge property and attribute states we end up with the encoding shown in Figure 4.20, but if we do merge than we end up with the encoding in Figure 4.21. These property states are merged because the invariable *waypoint* is shared. However, no cycles in the causal graph are broken by merging these states so no benefit is gained. Instead we blow up the size of the encoding, because the rovers might not be *potentially equivalent* which means that these nodes are fully grounded, so the ordering in which the soil and rock analysis are done and which rover performs the analysis all increase the size of the encoding.

If the lifted transition graphs are not merged, then for every soil and rock analysis we end up with $2 + 2n$ nodes, where n is the number of rovers. If they are merged then we end up with $2n^2 + 2n + 1$ nodes. So the encoding is considerably larger, which affects the planner.

4.4.2 Comparison with the Causal Graph heuristic

We compare our lifted causal graph heuristic with the causal graph heuristic. We do not expect to be able to solve more problem instances, because – as we have explained in Section 4.2 – we ignore dependencies between objects of the same type. In addition we do not use any pruning techniques or helpful actions as Fast Downward does. Merging lifted transition graphs allows us to extract more informative heuristics, but we do not

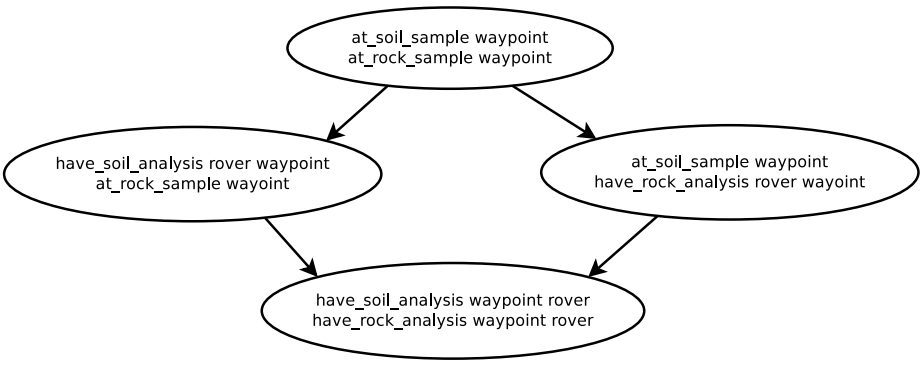

Figure 4.21: Merged lifted transition graphs for the soil and rock analysis property states.

expect this to offset the advantages Fast Downward has over our planner. On the other hand we expect to use far less memory to represent the lifted transition graphs and causal graph. This will lead to a reduced memory usage and allows us to use our planner on larger problem instances.

The number of states that are expanded are depicted in Figure 4.22 and the quality of the found solutions is depicted in Figure 4.23. As we can see we expand many more states than the causal graph heuristic; this is to be expected because Fast Downward utilises *preferred operators*, which are like *helpful actions* which prune the search space. We have not implemented such a pruning technique and because our heuristic is less informative than the causal graph heuristic we explore a larger part of the search space. We see good scaling behaviour for the Zeno, Driverlog, and Storage domains. However, the Blocksworld performs very poorly. This seems odd, because the lifted causal graph encoding is much smaller than the causal graph encoding. The reason why we require more memory than the causal graph heuristic is because we need to store more states while searching. This is due to a less informative heuristic and the lack of pruning in our heuristic. We present solutions to this problem when we present our conclusions.

4.4.3 Comparison with the Additive Enhanced Context heuristic

The additive enhanced context heuristic is a reformulation of the causal graph heuristic that does not require the causal graph to be acyclic. We expect that this reformulation will produce results that are very similar to the causal graph heuristic.

We see the same results when we compare our approach with the enhanced context heuristic. The number of states expanded is depicted in Figure 4.26 and the plan quality of the found solutions is depicted in Figure 4.27. The results are very similar to the causal graph heuristic.

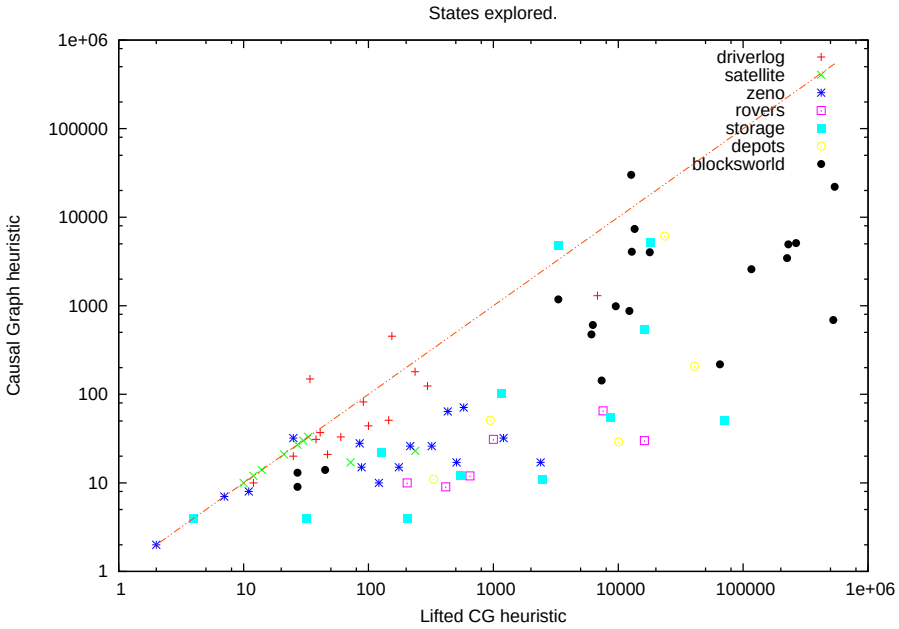

Figure 4.22: Merged lifted causal graph heuristic v.s. the causal graph heuristic.

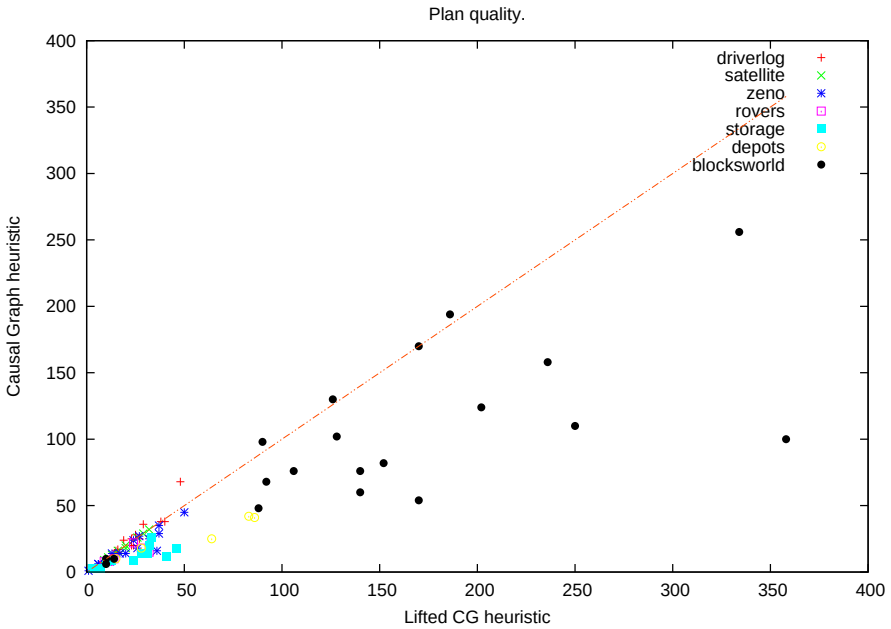

Figure 4.23: Merged lifted causal graph heuristic v.s. the causal graph heuristic.

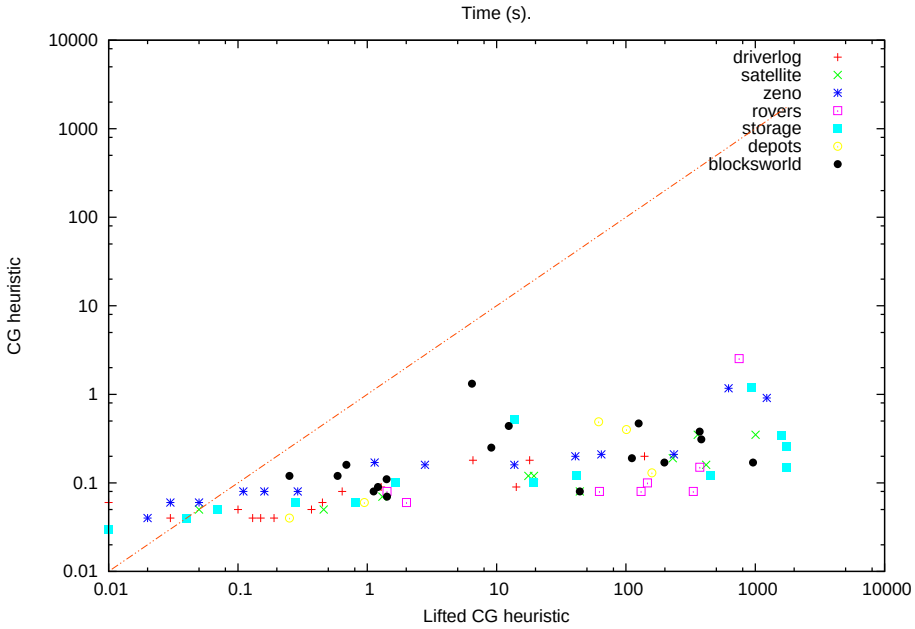

Figure 4.24: Merged lifted causal graph heuristic v.s. the causal graph heuristic.

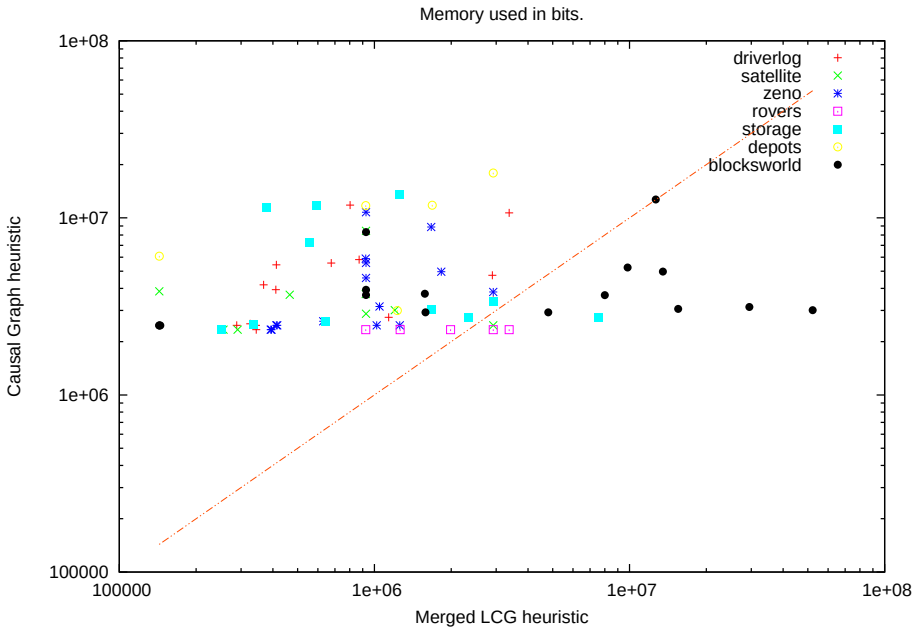

Figure 4.25: Merged lifted causal graph heuristic v.s. the causal graph heuristic.

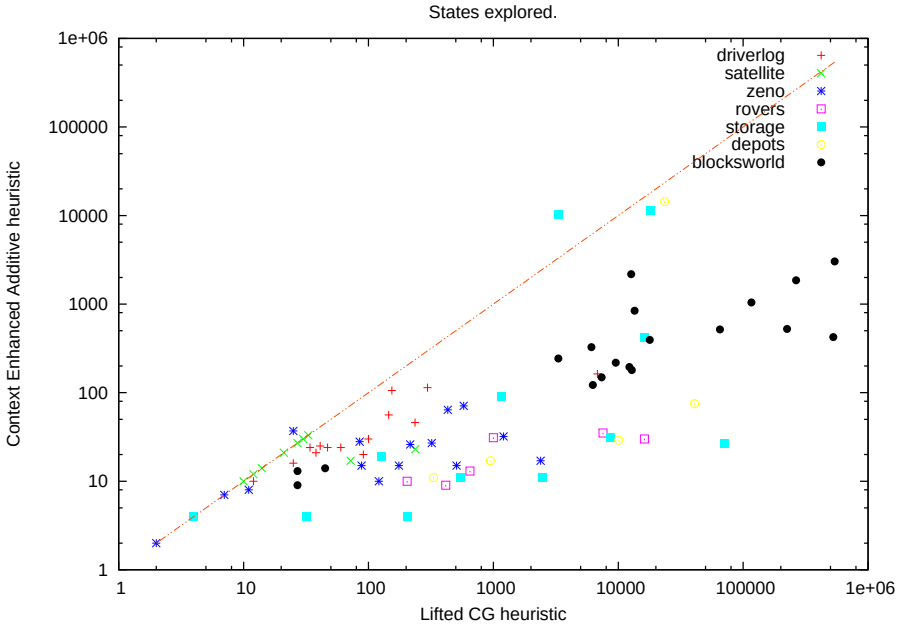

Figure 4.26: Merged lifted causal graph heuristic v.s. the context enhanced additive heuristic.

4.4.4 Compare to the Merge and Shrink heuristic

Lastly we compare our heuristic to the Merge and Shrink heuristic. We can set the number of states that are represented in this abstraction so the memory constraints should be fairly constant (except it still requires grounding to generate this abstraction). We have used the same configuration as has been used in [28] and we have used $N = 50,000$ as the number of states that can be stored in the merged *transition graph*. We use the same method to merge lifted transition graphs, but we do not require grounding to accomplish that. We expect to use less memory. We also expect that this algorithm is the weakest compared to the causal graph and additive causal graph heuristic, because it is admissible and introduces short cuts in the merge step (see Section 2.4.2). We also expect that this heuristic will not scale to larger planning problems; in this regard it shares the same weakness as pattern databases if we scale the problem instances large enough because N will stay the same while the number of states gets bigger.

The number of states expanded is depicted in Figure 4.30 and the plan quality of the found solutions is depicted in Figure 4.31. Although we solve more problems (see Table 4.2) we still explore more states for most planning problems. The merge and shrink heuristic has a near constant memory footprint because it limits the number of states that can be stored in the merged *transition graph*. For most planning domains we compare favourably, because we use less memory and manage to solve more problems than the merge and shrink heuristic.

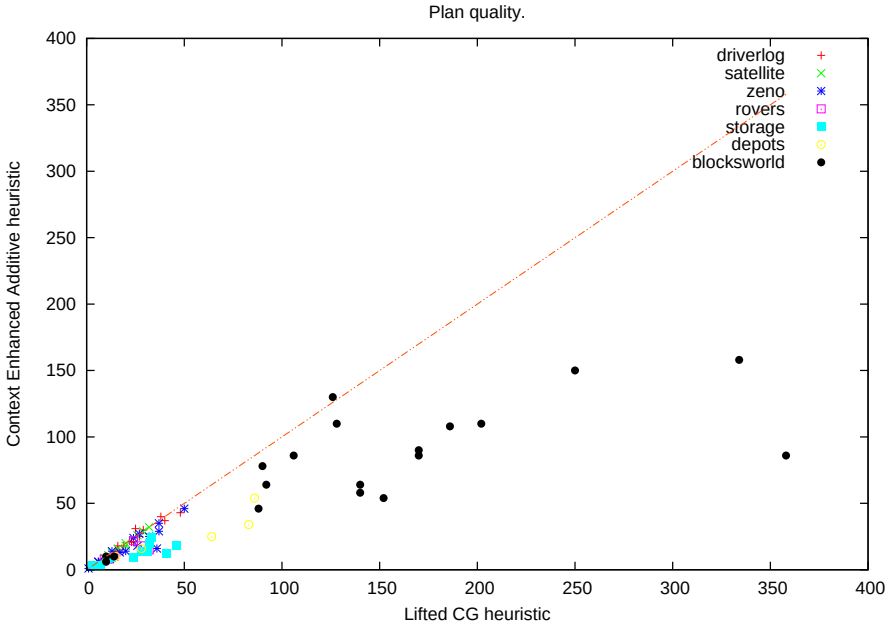

Figure 4.27: Merged lifted causal graph heuristic v.s. the context enhanced additive heuristic.

4.4.5 Summary

Due to the lack of pruning techniques our methods do not perform as well as the causal graph heuristic and context enhanced heuristic. The implementation of such techniques could form part of future work and is discussed in Section 5.1.3. The overview of the number of problems solved is depicted in Table 4.2. We see that we solve slightly more problems than the Merge and Shrink heuristic which also does not use any form of pruning. However, we also see that we perform very badly on the *Rovers* domain. The reason for this is discussed in Section 4.4.1.

Our heuristic performs reasonably well on the *Blocksworld* domain, which seems odd at first because we do not take any dependencies between blocks into account. However, since the lifted transition graph only consists of 5 nodes and the heuristic estimate is the shortest path in this graph we can calculate the heuristic estimate *very* quickly. Although not accurate it proves to be enough to find a solution. We can clearly see from Figure 4.22, Figure 4.26, and Figure 4.30 that our approach expands many more nodes than any of the other techniques and this effect is also visible in the plan quality of our solutions.

The *Depots* domain contains a similar dependency between crates which we also ignore. However, for this domain we find that we do not benefit from loosing heuristic information in favour of calculating the heuristic more quickly. For both these domains we present a possible solution in Section 5.1.5.

The results support our hypothesis that while our planner does not perform better

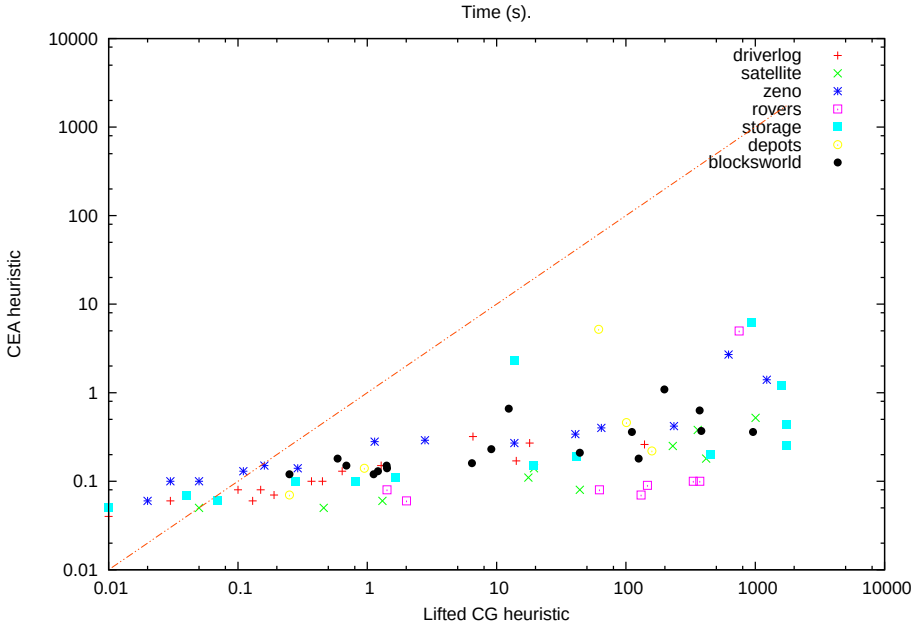

Figure 4.28: Merged lifted causal graph heuristic v.s. the context enhanced additive heuristic.

than the grounded heuristics (with the exception of the admissible merge and shrink heuristic) we do use far fewer memory to find solutions. We expect that the results above will improve significantly by implementing pruning techniques; this is left for future work and discussed in Section 5.1.3.

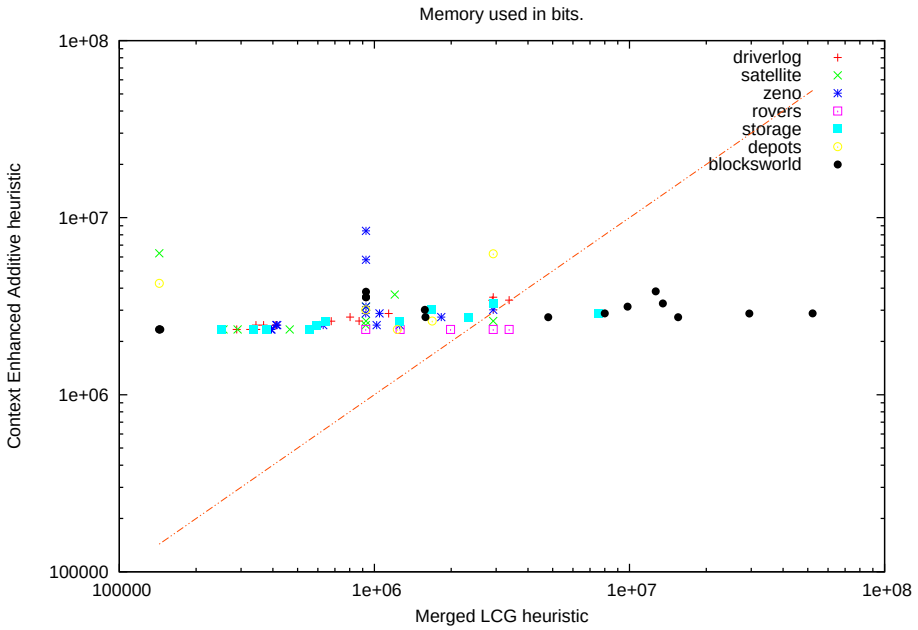

Figure 4.29: Merged lifted causal graph heuristic v.s. the context enhanced additive heuristic.

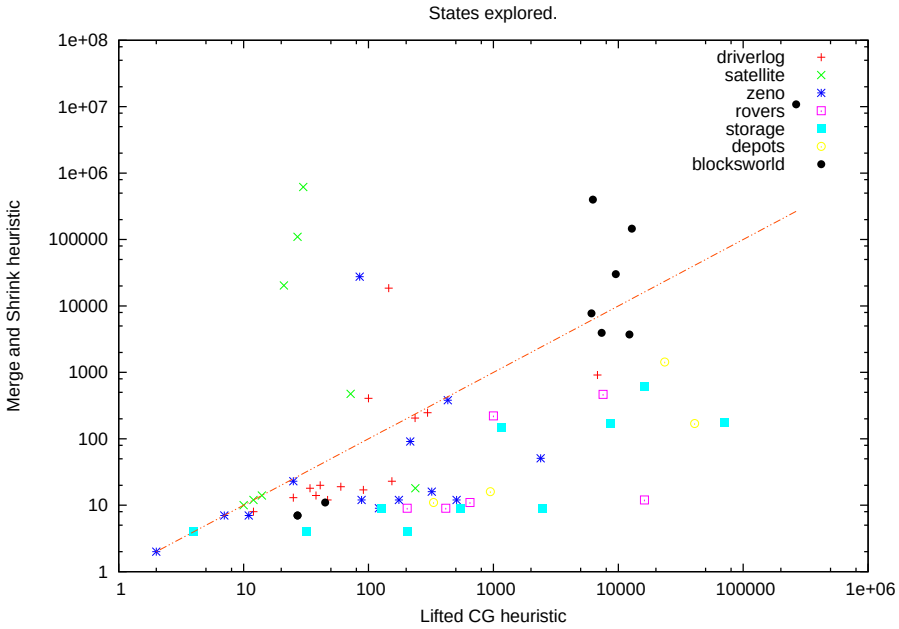

Figure 4.30: Merged lifted causal graph heuristic v.s. the Merge and Shrink heuristic.

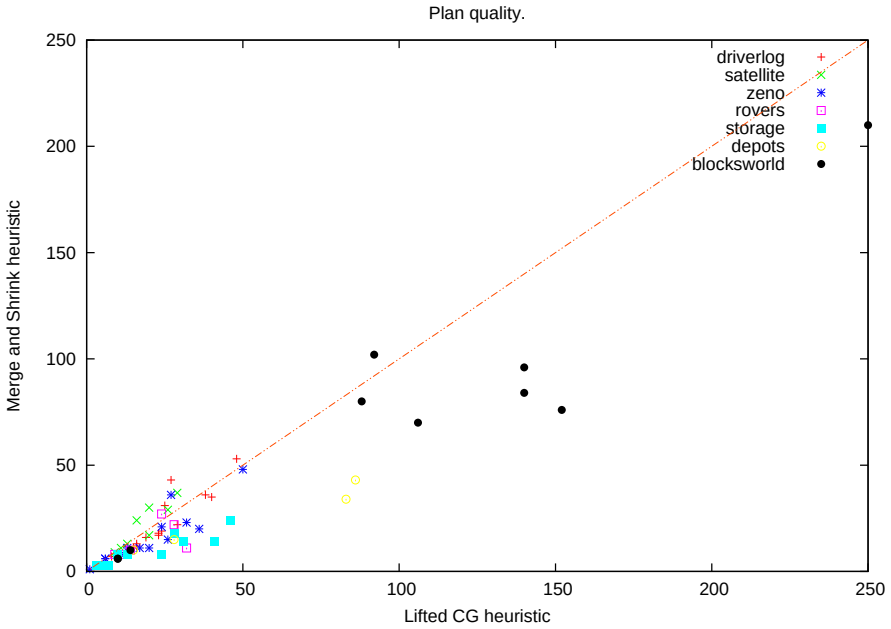

Figure 4.31: Merged lifted causal graph heuristic v.s. the Merge and Shrink heuristic.

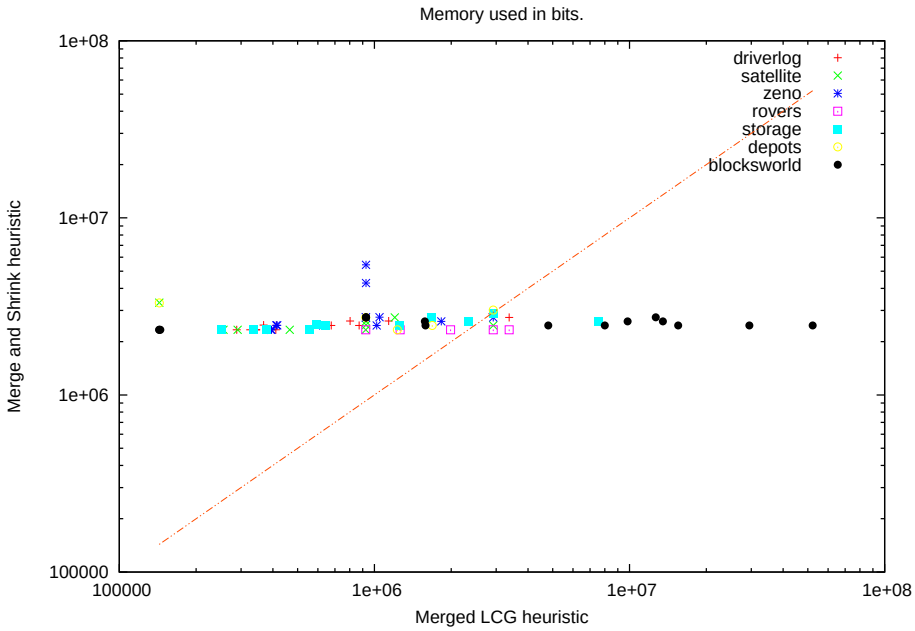

Figure 4.32: Merged lifted causal graph heuristic v.s. the Merge and Shrink heuristic.

Algorithm 6: The planning algorithm using h^{lcg}.

```
findPlan() P =search(TRUE);
if P = ∅ P =search(FALSE);
return P;
search(enable_ehc)
h_lowest = ∞;
priority_queue = {⟨s_0, {}, ∞⟩};
closed_list = {};
states_beforerestart = 1;
states_explored = 0;
last_beststate = s_0;
last_bestplan = {};
while priority_queue ≠ ∅ do
    ⟨s, P, h⟩ ∈ priority_queue | ¬∃⟨s', P', h'⟩ ∈ priority_queueh' < h;
    priority_queue = priority_queue \ ⟨s, P⟩;
    if s ∈ closed_list then
        continue;
    closed_list = closed_list ∪ s;
    if s_g ⊆ s then
        return P;
    for ⟨s', a⟩ ∈ successorsof s do
        h_s = h^lcg;
        if h_s ≡ ∞ then
            continue;
        states_explored = states_explored + 1;
        if h < h_lowest then
            states_explored = 0;
            last_beststate = s';
            last_bestplan = {P ∪ a};
            if enable_ehc then
                h_lowest = h;
                priority_queue = ∅;
        if enable_ehc AND states_explored ≡ states_beforerestart then
            priority_queue = {⟨last_beststate, last_bestplan⟩};
            states_beforerestart = states_beforerestart * 2;
        else
            priority_queue = priority_queue ∪ {⟨s', {P ∪ a}, h_s⟩};
return ∅;
```

Domain	h^{lcg} no merging	h^{lcg} allow merging
Driverlog	12	14
Zeno	14	15
Blocksworld	20	20
Storage	13	17
Depots	5	5
Satellite	9	9
Rovers	8	6
Total	81	86

Table 4.1: Number of problems solved.

Domain	h^{lcg} allow merging	h^{cg}	h^{cea}	h^{ms}
Driverlog	14	18	18	15
Zeno	15	19	20	13
Blocksworld	20	20	19	10
Storage	17	18	16	12
Depots	5	14	12	6
Satellite	9	19	19	12
Rovers	6	17	16	16
Total	86	125	120	84

Table 4.2: Number of problems solved.

Chapter 5

Conclusions and Future Work

In this thesis we have presented a new forward-chaining planner that uses two lifted heuristics: the lifted RPG heuristic and the lifted causal graph heuristic. We have shown that this planner uses less memory than the state-of-the-art planners such as Fast Forward and Fast Downward and is able to solve larger problem instances than any state-of-the-art planner can due to memory constraints (see Section 3.7.6).

For the lifted RPG we introduced new pruning techniques and discussed multiple configurations of this heuristic and shown their performance. Interestingly we found that the *fully lifted* compares comparably with the *partially lifted* configuration even though the latter produces better heuristic estimates and better quality plans. This is due to the fact that the former can find relaxed plans more quickly and is thus able to search more states than the latter.

The lifted causal graph heuristic uses techniques from Merge and Shrink to merge lifted transition graphs, which reduces lifted transition graphs and – by extension – the number of cycles in the causal graph. This novel technique allows us to retain more information that would otherwise be lost by breaking cycles in the causal graph. This yields better heuristic estimates. This technique is applicable to the causal graph heuristic. However, because DTGs are built per *object* instead of per *type* it needs a lot more memory to store these merged DTGs. If domains become large enough this technique would be infeasible for Fast Downward to deal with while we can still utilise this technique.

We have identified one problem with merging lifted transition graphs in the *Rover* domain. This led to an unnecessary blowup of the structures that does not benefit us at all. It would be simple enough to detect if there are any interactions between any two lifted transition graphs and if that is not the case we can forego merging these graphs. However, this is reserved for future work which we will discuss first.

5.1 Future Work

Before presenting the conclusions of this thesis we present future work.

	Naive	Enhanced Fully Lifted	Enhanced Partially Lifted	
Domain	default	h+p	h+p	Aggressive pruning
Driverlog	9	15	18	17
Zeno	6	15	14	15
Blocksworld	3	20	20	20
Storage	17	17	16	18
Depots	4	13	12	12
Satellite	7	17	18	19
Rovers	11	18	19	17
Total	57	115	117	118

Table 5.1: Number of problems solved. h means helpful actions enabled. p means that goals are preserved.

5.1.1 Lifted Landmarks

In Section 2.4.3 we discussed how landmarks can be extracted from a planning domain and how LAMA utilises them. We can find landmarks using the same techniques used by LAMA but on the lifted structures. The landmarks found by taking the conjunction of the preconditions of all goals that can achieve a landmark can be used without any problems. To find *first achievers* we can use the same technique we used to preserve goals when constructing the lifted transition graph. Similarly, we can search for nodes that need to be traversed for every path that exists between the value that is true in the current state and the value in the goal state for a state variable. Interestingly this would yield conjunctive landmarks like those produced by the AND-OR graph technique [35] but without having to construct a huge data structure to extract them.

Once we have a landmark-generation tree we can adapt the preserve goal pruning technique as discussed in Section 3.5.6. This way we can relax the constraints on constructing the lifted RPG until we find a graph from which we can extract a valid relaxed plan.

Alternatively we can use it in the same way as LAMA as part of the heuristic value on top of our heuristic estimates we have presented in this thesis.

5.1.2 Pruning

We have implemented both novel and already existing pruning techniques into our planner, but there are more that are worth exploring. Most notable is the pruning technique used by FF, where it prunes the search space by committing to any state that has a better heuristic than any state expanded till that point. We have implemented an initial version of that technique and have depicted the results in Figure 5.1 and Figure 5.2.

We observe that we have to expand less states than before, although the plan quality remains unaffected. The number of planning problems solved is depicted in Table 5.1; the results are slightly better than those for the previous configurations.

Our planning system has not implemented any pruning techniques for the lifted causal graph heuristic. Fast Downward uses a variation of helpful actions to prune

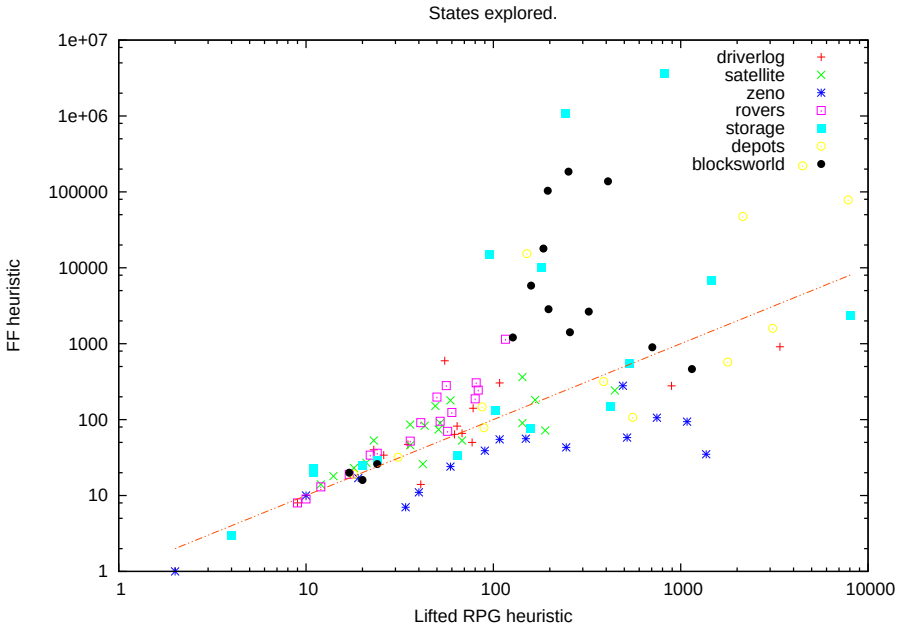

Figure 5.1: Results by more aggressive pruning.

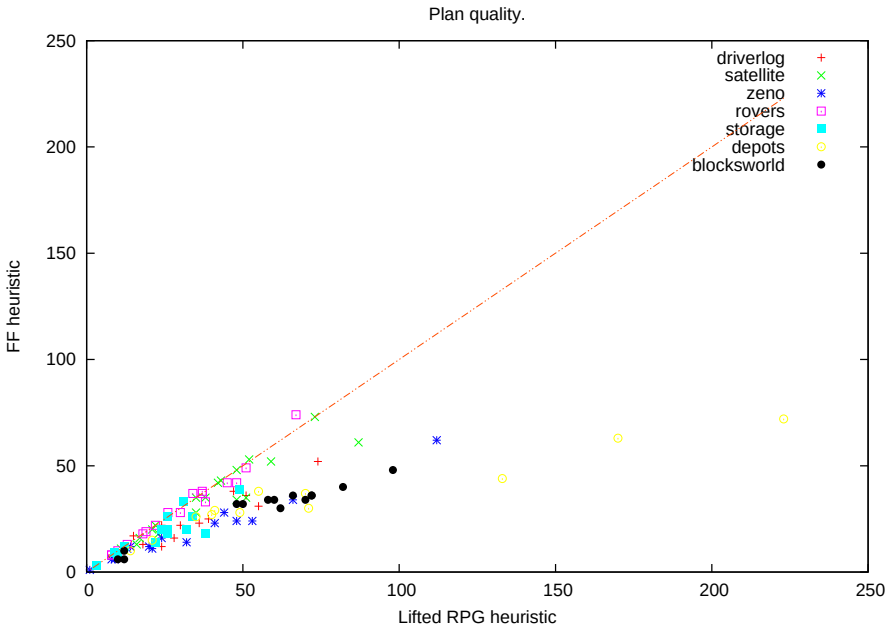

Figure 5.2: Results of more aggressive pruning.

the search space. We think that technique can be used in conjunction with the lifted transition graph heuristic and will help to solve larger problem instances.

5.1.3 Helpful transitions

In our implementation we have not used any pruning techniques, which is reflected in the results we obtained from our experiments as discussed in Section 4.4. We can improve our results by pruning the search space by only considering the preferred operators. We expect that this would yield the same improvements as noted in Section 3.7 with the implementation of helpful actions in the lifted causal graph heuristics.

Given a typed multi-valued planning task and the set of abstracted *lifted transition graphs* (preconditions and effects are removed based on the way the cycles are broken in the causal graph) then the set of helpful actions consists of each first action in $\pi_g \mid g \in s_g$ that is applicable in s_0, where π_g is the plan found to solve the goal g. Unlike the technique we use to extract *helpful actions* from the *lifted relaxed planning graphs* there is no guarantee that we can actually find any *helpful transitions*. This is because of the way cycles are removed from the causal graph; a set of preconditions might be removed in such a way that none of the transitions in the relaxed plan are applicable in the initial state.

5.1.4 Alternative search techniques

In general we can conclude that if we want to solve bigger problem instances then we need a planner with the following properties:

- An informative heuristic that does not need the entire planning problem to be grounded.
- Strong pruning techniques (not necessarily solution preserving) that effectively reduces the search space.
- A search technique that efficiently navigates the search space.

We have presented a forward-chaining planner in this thesis with these properties. However it would be interesting to see how the pruning techniques and heuristics would perform in a least commitment planner like *partial-order planners*. Previous results with RePOP [42] have shown that heuristics developed for forward-chaining planners translate well to partial-order planners. The paper on VHPOP [57] comments that it would be interesting to see how the FF heuristic performs in the context of partial-order planners. This line of investigation remains part of future work.

5.1.5 Recursion

In Section 4 we discussed the lifted causal graph heuristic. One of the weaknesses of this heuristic is that it ignores any dependencies between state variables of the same *type*, most notably for *crates* in the Depots domains and *blocks* in the Blocksworld domain. What is interesting about the dependencies between these types is that we can

rewrite the transition of the *lifted transition graphs* to expose the recursive dependencies. For example, consider the transition *(unstack block block')* between the nodes

$$\{(on\ block\ block'), (on\ block'\ block'')\}$$

and

$$\{(clear\ block'), (on\ block'\ block'')\}.$$

In our encoding we ignore the precondition *(clear block)* because this is a value of the same state variable. By ignoring this precondition we can unstack any block from another block even if there are other blocks on top of the other block. This affects the heuristic estimate quite significantly. We can rewrite this precondition as a recursive function $C(block) = (clear\ block) \vee ((on\ block'''\ block) \wedge C(block'''))$ and count the number of necessary calls to this function towards the cost of achieving the precondition *(clear block)*. This should improve the heuristic value returned for both the *Blocksworld* and *Depots* problem instances.

5.2 Conclusions

In this thesis we have addressed a problem that – thus far – has been generally ignored by the state-of-the-art research in planning, grounding. We have explored the necessity of grounding by exploring the limits it artificially imposes on planning systems. Planning systems that rely on grounding scale very badly in terms of memory as the size of planning problems grow, it can also take a substantial amount of time to ground a planning problem. We have shown that research into partial-order planners has produced lifted planning systems that use heuristics that do not require grounding, but the informativeness of these heuristics is very poor compared to the state-of-the-art.

The problem of building a planning system that can solve very large problem instances is still an open question. However, in this thesis we have investigated whether grounding is necessary to produce informative heuristics estimates. By doing so we have addressed one part of this problem by removing the dependency on grounding to calculate informative heuristic estimates. This allows our planning system to encode, and start planning on, on large problem instances and in some cases even find plans for large problem instances that are currently unsolvable by state-of-the-art planners. However, while we can start planning on much larger instances than can even be grounded using state of the art planning systems, in general we are unable to solve very large problem instances because lifted heuristics are insufficiently informative. Extracting heuristics from lifted plan graphs results in heuristics that are necessarily less informative than those extracted from grounded plan graphs. There are several developments that we expect to improve this picture, which are topics for future work.

The planning system developed in this thesis can utilise the two novel heuristics that we have introduced.

1. Lifted Relaxed Planning Graph Heuristic: This heuristic is based on the FF heuristic and exploits equivalent relationships between objects. We have shown

that the *naive* version of this heuristic is very uninformative, but can be greatly improved by accounting for *substitutions*. We have shown that his heuristic can be further improved by *goal preservation* and by pruning the search space by utilising helpful actions.

2. Lifted Causal Graph Heuristic: The second heuristics is based on the CG heuristic and – like the previous heuristic – exploits object equivalence relations. We construct lifted variants of the domain transition graphs and relax the causal relationships between objects that are part of the same equivalent class. This allows for a very compact abstraction; we further compressed these data structures by *merging* lifted domain transition graphs that apply to the same equivalent classes. This reduces the number of cycles in the causal graph which means that we have to break less cycles which leads to better heuristic estimates.

We have shown that these lifted heuristics are very informative, compared to their grounded counterparts, and require substantially less memory to compute. Furthermore, both lifted heuristics can be calculated quicker than their grounded counterparts which means that the usual dichotomy of trading memory for time does not apply here; we calculate heuristics quicker *and* use less memory. This is an important result and we expect that future research – parts of which are listed in Section 5.1 – will enable planning systems to become better scalable and be able to solve far larger problem instances than the current state-of-the-art.

The focus of this thesis has been the formulation of new heuristics that do not require grounding. We hope that object equivalent relationships can be exploited outside the scope of heuristic calculations and prove to be a useful tool to create planning systems that scale better. An interesting research direction would be to revisit partial-order planners and see how this planning strategy benefits from techniques introduced in this thesis. This planning strategy is a likely candidate when talking about lifted planning systems since partial-ordered planners are inherently lifted and might present more opportunities for symmetries to be exploited.

We hope that this thesis has convinced the reader that the heuristics and pruning techniques presented in this paper make it possible to solve larger problem instances than is currently possible due to (1) memory constraints and (2) the dependency on grounding by state-of-the-art planners. We believe that this work is a good starting point for future investigation into least commitment planners and planning systems designed to solve large problem instances that cannot be solved currently due to their size. This will allow planning technology to be integrated with applications that deal with a large number of objects. For example, large logistic problems, warehouse applications where robots need to plan where to store / retrieve packages, supply chain management, and many others.

Bibliography

[1] F. Bacchus and Q. Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 71:43–100, 1993.

[2] S. Bernardini and D. E. Smith. Automatic synthesis of temporal invariants. In *Symposium on Abstraction, Reformulation, and Approximation*. AAAI, 2011.

[3] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:1636–1642, 1995.

[4] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129:5–33, 2001.

[5] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:165–204, 1994.

[6] A. Coles and A. Smith. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research*, 28:119–156, 2007.

[7] S. Edelkamp. Symbolic pattern databases in heuristic search planning. In *Artificial Intelligence Planning Systems*, pages 274–283, 2002.

[8] S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In *European Conference on Planning*, pages 135–147. Springer, 1999.

[9] S. Edelkamp and M. Helmert. On the implementation of MIPS. In *Proceedings of Artificial Intelligence Planning Systems - Workshop on Model Theoretic Approaches to Planning*, pages 18–25. AAAI press, 2000.

[10] E. Emerson, A. Sistla, and H. Weyl. Symmetry and model checking. In *Formal Methods in System Design*, volume 9, pages 105–131. Kluwer Academic Publishers, 1994.

[11] K. Erol, D. S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning: A detailed analysis. *Artificial Intelligence*, 76:75–88, 1991.

[12] A. Felner, R. Korf, and S. Hanan. Disjoint pattern database heuristics. *Artificial Intelligence*, 134:9–22, 2002.

[13] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. In *International Joint Conference on Artificial Intelligence*, pages 608–620, 1971.

[14] J. E. Flórez, Á. T. A. de Reyna, J. García, C. Linares López, A. G. Olaya, and D. Borrajo. Planning multi-modal transportation problems. In *International Conference on Automated Planning and Scheduling*. AAAI, 2011.

[15] M. Fox and D. Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367–421, 1998.

[16] M. Fox and D. Long. The detection and exploitation of symmetry in planning problems. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 956–961, 1999.

[17] M. Fox and D. Long. Hybrid STAN: Identifying and managing combinatorial optimisation sub-problems in planning. In *International Joint Conference on Artificial Intelligence-01*, pages 445–452. Morgan Kaufmann, 2000.

[18] M. Fox and D. Long. Extending the exploitation of symmetries in planning. In *Artificial Intelligence Planning Systems*, pages 83–91, 2002.

[19] A. Gerevini, A. Saetti, and I. Serina. On managing temporal information for handling durative actions in LPG. In *AI*IA*, volume 2829 of *Lecture Notes in Computer Science*, pages 91–104. Springer, 2003.

[20] A. Gerevini and L. Schubert. Accelerating partial-order planners: Some techniques for effective search control and pruning. *Journal of Artificial Intelligence Research*, 5:95–137, 1996.

[21] A. Gerevini and L. Schubert. Inferring state constraints for domain-independent planning. In *Conference on Innovative Applications of Artificial Intelligence*, pages 905–912. AAAI Press, 1998.

[22] M. Ghallab, C. K. Isi, S. Penberthy, D. E. Smith, Y. Sun, and D. Weld. PDDL - the planning domain definition language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[23] O. Giménez and A. Jonsson. Planning over chain causal graphs for variables with domains of size 5 is NP-Hard. *Journal Artificial Intelligence Research*, 34:675–706, 2009.

[24] P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In *Conference on Artificial Intelligence Planning Systems*, pages 140–149. AAAI Press, 2000.

[25] M. Helmert. A planning heuristic based on causal graph analysis. *International Conference on Automated Planning and Scheduling*, pages 161–170, 2004.

[26] M. Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[27] M. Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

[28] M. Helmert, Albert-Ludwigs-Universität Freiburg, P. Haslum, and J. Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *International Conference on Automated Planning and Scheduling*, pages 176–183, 2007.

[29] M. Helmert and G. Röger. How good is almost perfect. In *International Conference on Automated Planning and Scheduling-Workshop on Heuristics for Domain-Independent Planning*, 2007.

[30] J. Hoffmann. FF: The fast-forward planning system. *AI magazine*, 22:57–62, 2001.

[31] J. Hoffmann and S. Edelkamp. The deterministic part of IPC-4: an overview. *Journal of Artificial Intelligence Research*, 24:519–579, October 2005.

[32] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14, 2001.

[33] J. Hoffmann, J. Porteous, and L. Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

[34] R. Howey, D. Long, and M. Fox. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *International Conference on Tools with Artificial Intelligence*, pages 294–301, 2004.

[35] E. Keyder, S. Richter, and M. Helmert. Sound and complete landmarks for and/or graphs. In *European Conference on Artificial Intelligence*, pages 335–340. IOS Press, 2010.

[36] C. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68:243–302, 1994.

[37] J. Koehler and J. Hoffmann. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal Artificial Intellicence Research*, 12(1):339–386, June 2000.

[38] R. E. Korf. Finding optimal solutions to Rubik's cube using pattern databases. Conference on Innovative Applications of Artificial Intelligence, pages 700–705. AAAI Press, 1997.

[39] D. Long and M. Fox. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.

[40] E. Luks. *Permutation Groups and Polynomial Time Computation*. Princeton University Press, 2007.

[41] B. D. McKay. NAUTY user's guide (version 1.5). Technical Report TR-CS-90-02, Australian National University, Department of Computer Science, 1990.

[42] X. Nguyen and S. Kambhampati. Reviving partial order planning. volume 1 of *International Joint Conference on Artificial Intelligence*, pages 459–464, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[43] E. P. D. Pednault. ADL: exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pages 324–332, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[44] J. S. Penberthy and D. S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Principles of Knowledge Representation and Reasoning*, pages 103–114. Morgan Kaufmann, 1992.

[45] C. Piacentini, V. Alimisis, M. Fox, and D. Long. Combining a temporal planner with an external solver for the power balancing problem in an electricity network. In *International Conference on Automated Planning and Scheduling*, 2013.

[46] N. Pochter, A. Zohar, and J. S. Rosenschein. Exploiting problem symmetries in state-based planners. In *Association for the Advancement of Artificial Intelligence*, pages 1004–1009, 2011.

[47] N. Pochter, A. Zohar, and J. S. Rosenschein. Exploiting problem symmetries in state-based planners. In *The Twenty-Fifth National Conference on Artificial Intelligence*, pages 1004–1009, San Francisco, August 2011.

[48] M. E. Pollack, D. Joslin, and M. Paolucci. Flaw selection strategies for partial-order planning. *Journal of Artificial Intelligence Research*, 6:223–262, 1997.

[49] J. Porteous, D. Long, and M. Fox. The identification and exploitation of almost symmetry in planning problems. In *UK Planning And Scheduling Special Interest Group*, 2004.

[50] J. Porteous and L. Sebastia. Extracting and ordering landmarks for planning. *Journal of Artificial Intelligence Research*, 22:2004, 2000.

[51] J. Porteous, L. Sebastia, and J. Hoffmann. On the extraction, ordering, and usage of landmarks in planning. In *Proceedings European Conference on Planning*, pages 37–48, 2001.

[52] J. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Methodologies for Intelligent Systems*, pages 350–361, London, UK, UK, 1993. Springer-Verlag.

[53] S. Richter, M. Helmert, and M. Westphal. Landmarks revisited. In *Association for the Advancement of Artificial Intelligence*, pages 975–982. AAAI Press, 2008.

[54] S. Richter and M. Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. In *Journal of Artificial Intelligence Research*, volume 39, pages 127–177, 2010.

[55] J. Rintanen. Symmetry reduction for SAT representations of transition systems. In *International Conference on Automated Planning and Scheduling*, pages 32–41. AAAI, 2003.

[56] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. In *International Joint Conference on Artificial Intelligence*, pages 412–422, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[57] D. E. Smith, J. Frank, and A. K. Jónsson. Bridging the gap between planning and scheduling. *The Knowledge Engineering Review*, 15(1):47–83, 2000.

[58] B. Srivastava. Realplan: Decoupling causal and resource reasoning in planning. In *Conference on Innovative Applications of Artificial Intelligence*, pages 812–818. AAAI/MIT Press, 2000.

[59] A. Tate. Project planning using a hierarchical non-linear planner. Technical Report 25, Dept. of Artificial Intelligence, Edinburgh University, 1977.

[60] J. D. Tenenberg. *Abstraction in planning*. PhD thesis, Rochester, NY, USA, 1988. Order No: GAX88-16885.

[61] T. Walsh. Breaking value symmetry. In *Principles and Practice of Constraint Programming*, pages 880–887, 2007.

[62] H. L. S. Younes and R. G. Simmons. VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research*, 20:405–430, 2003.